2

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

EVALUATIONS OF SOME SCHEDULING ALGORITHMS FOR HARD REAL-TIME SYSTEMS

by

Bao Hua Fan

June 1990

Thesis Advisor: Valids Berzins

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable) CS	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Evaluations of some scheduling algorithms for Hard Real-Time Systems

12. PERSONAL AUTHOR(S)
Bao-Hua, Fan

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM 08/89 TO 06/90	June 1990	125

16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Hard Real-Time Systems, Prototype System Description Language (PSDL), Execution Support System (ESS), Computer Aided Prototyping System(CAPS)

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The Computer Aided Prototyping System (CAPS) and the Prototype System Description Language (PSDL) represent a pioneering effort in the field of software development. Execution Support System (ESS) within the framework of CAPS. The Static Scheduler is one of the critical elements of the ESS which extracts critical timing constraints and precedence constraints for operators and schedules the time-critical operators to guarantee that their timing constraints will be met. The Static Scheduler uses the information of timing constraints and precedence constraints to determine whether a feasible schedule can be built. This construction provides the foundation for handling the execution for Real-Time systems. The goal of this thesis is to provide improved versions of the Static Scheduler.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Valids Berzins	22b. TELEPHONE (Include Area Code) (408) 646-2461; 22c. OFFICE SYMBOL CS/Be

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

Evaluations of Some Scheduling Algorithms for Hard Real-Time Systems

by

Bao-Hua Fan
Commander, R.O.C Navy
B.S., Chinese Naval Academy, 1977

Submitted in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1990

Author: Fan Bao Hua
Bao-Hua Fan

Approved by: Valdis Berzins
Valdis Berzins, Thesis Advisor

[illegible]
Man-Tak Shing, Second Reader

Robert B. McGhee
Robert B. McGhee, Chairman, Department of Computer Science

ABSTRACT

The Computer Aided Prototyping System (CAPS) and the Prototype System Description Language (PSDL) represent a pioneering effort in the field of software development. Execution of the prototype is controlled by an Execution Support System (ESS) within the framework of CAPS. The Static Scheduler is one of the critical elements of the ESS which extracts critical timing constraints and precedence constraints for operators and schedules the time-critical operators to guarantee that their timing constraints will be met. The Static Scheduler uses the information of timing constraints and precedence constraints to determine whether a feasible schedule can be built. This construction provides the foundation for handling the execution for hard real-time systems. The goal of this thesis is to provide improved versions of the Static Scheduler.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

A. SOFTWARE ENGINEERING

Software engineering is the application of scientific and mathematical principles by which the capabilities of computers are made useful through the application of computer software programs, procedures and related documentation. The goal of the software development is to create modifiable, efficient, reliable, and understandable software systems. System performance is important to the user. A delivered software system must accurately represent the user's stated requirements and also consistently produce highly reliable responses in the anticipated environment. Quality of the design is becoming increasingly important to both the user and the systems engineer. Software engineering encourages the use of a development life cycle methodology that systematically and consistently incorporates these goals and principles in the creation of software systems.

Computer software is information that exists in two basic forms: non-machine-executable components and machine-executable components. Figure 1 on page 2 illustrates the manner in which software is translated into machine-executable form. The software design is translated into a language form that specifies software data structure, procedural attributes, and related requirements. The language form is processed by a translator that converts it into machine-executable instructions.

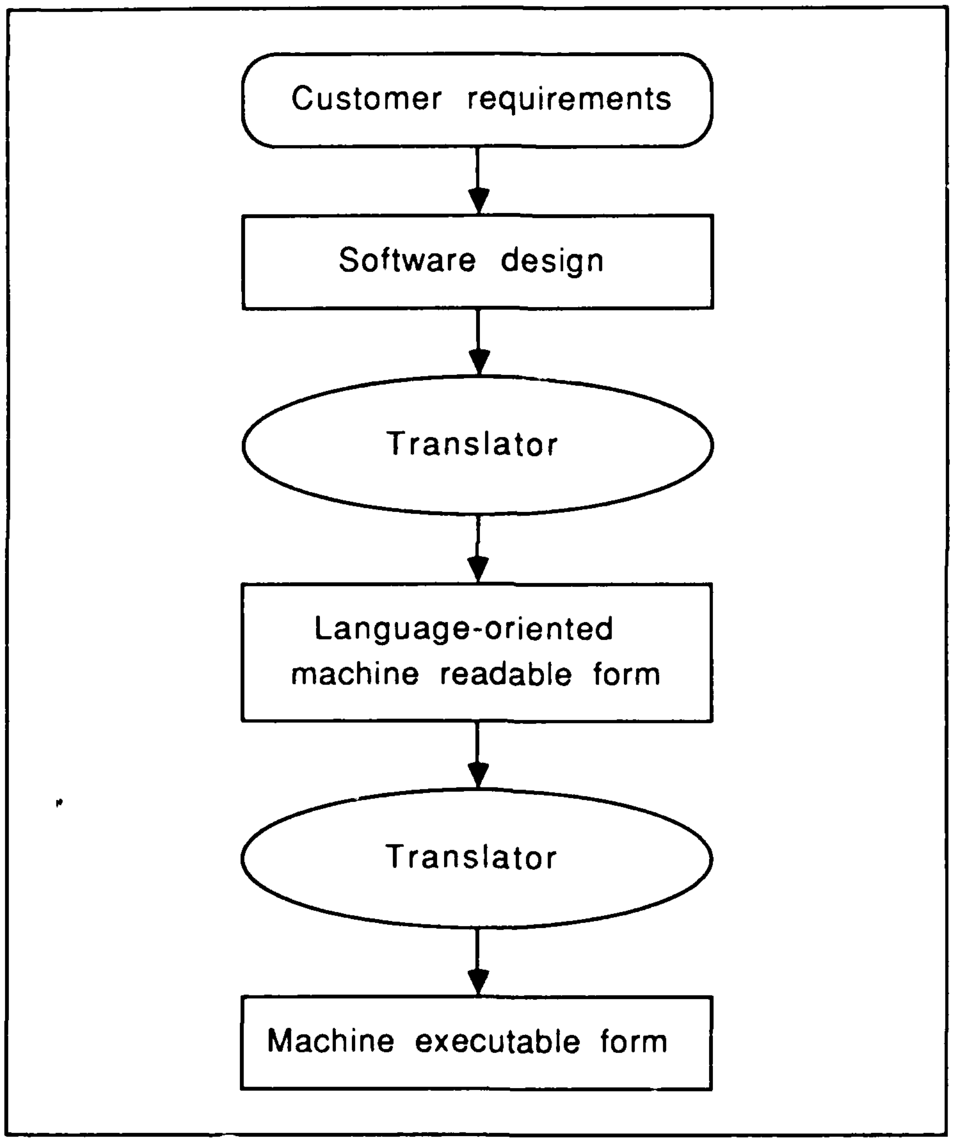


Figure 1 Software translation steps

The traditional software life cycle and rapid prototyping are two of the more common design methodologies used to maintain a scientific approach to software engineering.

1. Traditional Life Cycle

The traditional software cycle is based on the waterfall life cycle, which incorporates individual development stages. Figure 2 on page 3 shows a model of this cycle.

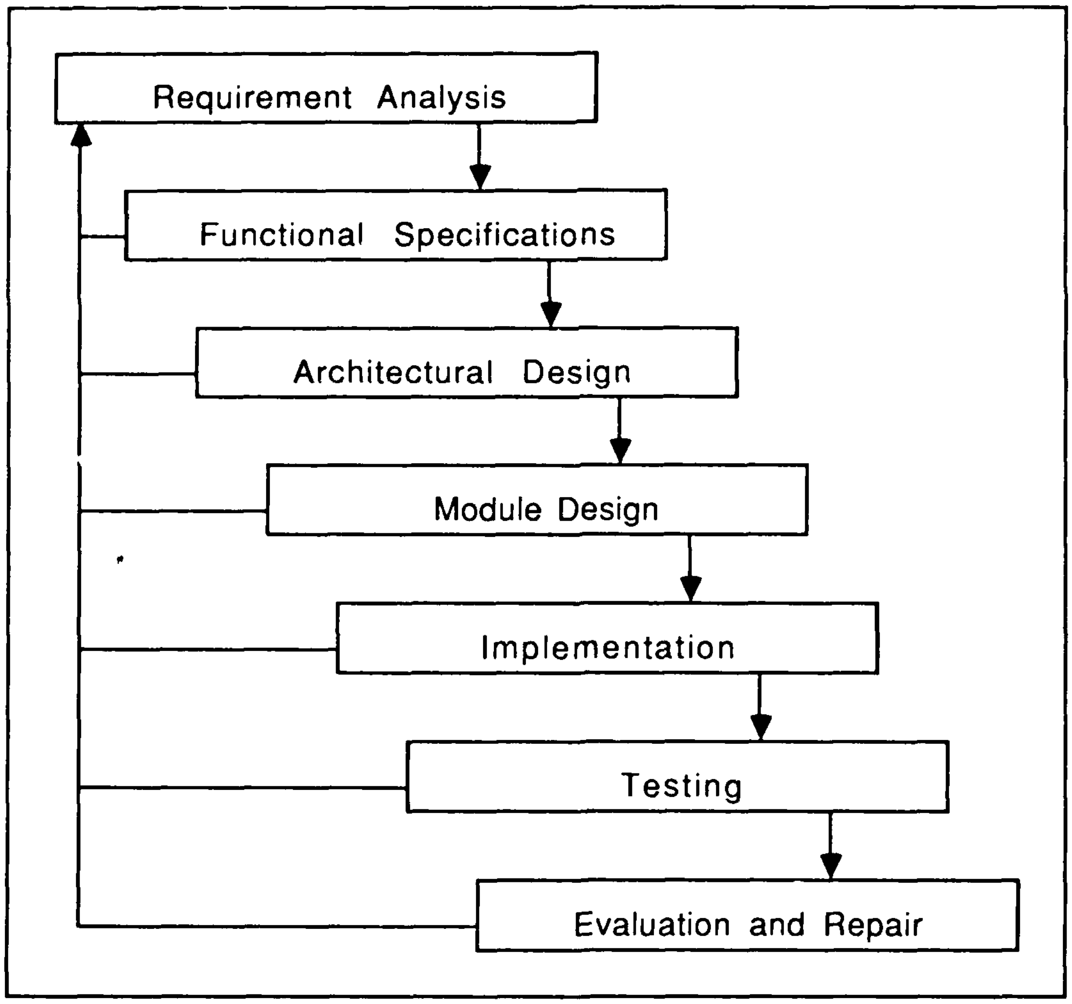


Figure 2 The Traditional Software Life Cycle Methodology

These phases are described as follows:

a. Requirements Analysis: This phase establishes the purpose of the proposed software system.

b. Functional Specifications: A model of the proposed system is constructed. This model contains only those aspects of the system that are visible to the users.

c. Architectural Design: A model of the implementation is constructed. The software modules and interfaces that will be used to realize the system are identified.

d. Module Design: During this phase of development, the algorithms and data structures to realize the behavior specified in the architectural design are chosen.

e. Implementation: Executable programs are produced, usually in a high level programming language.

f. Testing: In this phase, faults are detected by running programs with selected input data.

g. Evolution and Repair: New features/capabilities are added onto the system and necessary design changes are made to repair faults.

2. Rapid Prototyping

The methodology called rapid prototyping [Ref. 24] is proving to be much more efficient than the traditional life cycle in the design of large real-time systems. Under the rapid prototyping paradigm, an effort is made to ensure that the customer and the developer both understand what the customer's requirements for a software system are. The rapid prototyping methodology is made up of two phases: rapid prototyping and automatic program generation. A prototype is an executable model of the intended system and is the product of the rapid prototyping phase.

The prototype is only a partial representation of the intended system and includes only the system's most critical aspects [Ref. 34]. Figure 3 on page 6 illustrates the model of this approach [Ref. 24].

The user and the designer work together to define the requirements and specifications for the critical parts of the envisioned system. The designer then constructs a model or prototype of the system in a prototype description language. This model is defined at the specification level. The resulting prototype is a partial representation of the system, including only those attributes necessary for meeting the requirements. It serves as an aid in analysis and design rather than as production software.

During demonstrations of the prototype, the user evaluates the prototype's actual behavior against its expected behavior. If the prototype fails to execute properly, the user identifies problems and works with the designer to redefine the requirements. This process continues until the user determines that the prototype successfully captures the critical aspects of the envisioned system.

The designer uses the validated requirements as a basis for designing the production software. Additional work is often needed to construct a production version of the system. Experience with production use of a delivered system often leads to new customer goals, triggering further iterations of the prototyping cycle.

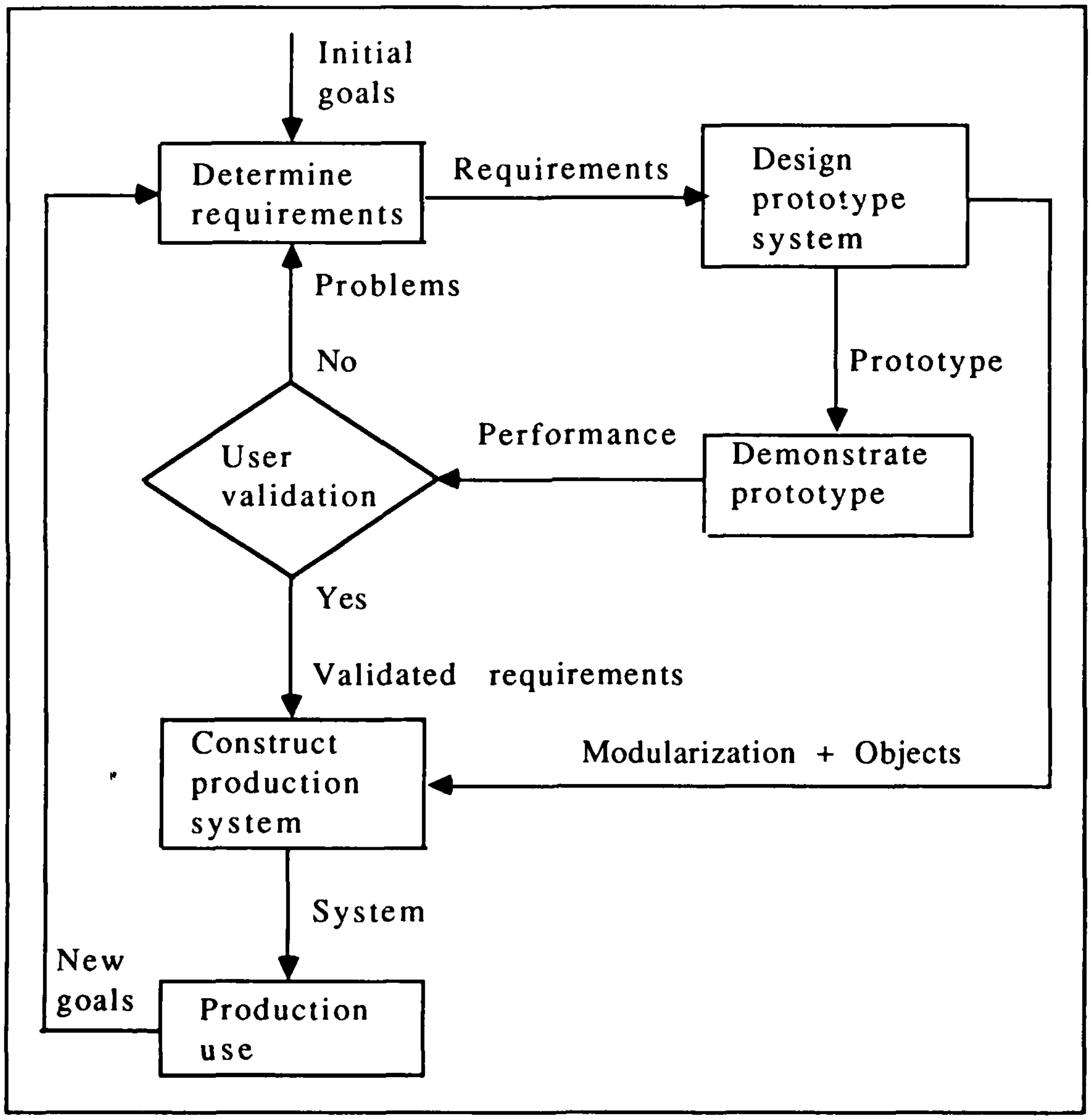


Figure 3 The prototyping cycle

3. Comparison of The Two Approaches

The traditional model of software development relies on the assumption that designers can stabilize and freeze the requirements. In practice, however, the design of accurate and stable requirements cannot be completed until users gain some experience with the proposed software system. Thus, requirements often must change after the initial implementation.

In traditional approaches, these requirements changes trigger changes to the production version of the system during the maintenance phase. In prototyping approaches, an appreciable fraction of the requirements changes trigger changes in a prototype version of the system. This is useful because a prototype description 1) is significantly simpler than the production code, 2) is expressed in a notation tailored to support modifications, and 3) is suitable for processing by software tools in a computer-aided prototyping environment.

These factors make it possible to modify a prototype more easily than a production version of the system. They make prototyping especially attractive for unfamiliar application areas with uncertain requirements. [Ref. 28]

B. REAL-TIME SYSTEMS

The ever-increasing use of computer systems is a clear evidence that the functional capabilities provided by them can be used very effectively for a variety of purposes and in a large number of fields. In many of these applications, the performance of the computer system is measured with metrics such as response time or turnaround time, the implication being that the faster the better, with no specific requirement being placed on the timing behavior of the system. Real-time applications are different from this paradigm of computation in that they impose strict requirements on the timing behavior of the system. The systems that support the execution of real-time applications and ensure that the timing requirements are met are often referred to as real-time systems. [Ref. 18]

There are two types of real-time systems, namely, soft real-time systems and hard real-time systems. In soft real-time systems, tasks are performed by the system as fast as possible, but they are not constrained to finish by specific times. On the other hand, in hard real-time systems, tasks have to be performed not only correctly, but also in a timely fashion. Otherwise, there might be severe consequences. Typically, a hard real-time task is characterized by its timing constraints, precedence constraints, and resource requirements. Flight control, automated manufacturing plants, telecommunications, and command and control systems are examples of such systems. [Ref. 11]

Hard real-time systems are characterized by the fact that severe consequences will result if the timing as well as the logical correctness properties of the system are not satisfied. Typically a hard real-time software system is a controlling system. The controlled system can be viewed as the environment with which the computer interacts.

Task scheduling in hard real-time systems can be static or dynamic. A static approach calculates schedules for tasks off-line and requires complete prior knowledge of tasks' characteristics. A dynamic approach determines schedules for tasks on the fly and allows tasks to be dynamically invoked. Although static approaches have low run-time cost, they are inflexible and cannot adapt to a changing environment or to an environment whose behavior is not completely predictable. When new tasks are added to a static system, the schedule for the entire system must be recalculated, which is expensive in terms of time and money. In contrast, dynamic approaches involve higher run-time costs, but, because of the way they are designed, they are flexible and can easily adapt to changes in the environment.

In summary, hard real-time systems differ from traditional systems in that deadlines or other explicit timing constraints are attached to tasks. The systems are in a position to make compromises, and faults, including timing faults, may cause catastrophic consequences. This implies that, unlike many systems where there is a separation between correctness and performance, in a hard real-time system correctness and performance are very tightly interrelated. Thus hard real-time systems solve the problem of missing deadlines in ways specific to the requirements of the target application.

C. THE STATIC SCHEDULER

One of the most critical components of the Computer Aided Prototyping System (CAPS) is the Static Scheduler subsystem of the Execution Support System (ESS). The static scheduler builds a static schedule for the execution of a prototype developed from the Prototype System Description Language (PSDL). The prototype consists of a set of tasks that must obey timing constraints and precedence relationships. The schedule proceduced by the static scheduler gives the precise execution order and timing of operators with hard real-time constraints in such a manner that all timing constraints are guaranteed to be met.

D. OBJECTIVES

The objective of this thesis is the implementation and evaluation of several schedulers that use different scheduling algorithms to find feasible schedules for the real-time prototypes satisfying the critical timing constraints and precedence relationships among operators in the prototype.

E. ORGANIZATION

Chapter II describes the previous research in hard real-time scheduling algorithms. It includes a discussion of the Computer Aided Prototype System (CAPS) and the Prototype System Description Language (PSDL). This chapter also presents a survey of static scheduling algorithms for a single processor environment. Chapter III designs a branch and bound algorithm for static scheduling. Chapter IV describes the details of the implementation. Chapter V presents conclusions and recommendations for the future work.

II. PREVIOUS RESEARCH AND SURVEY OF STATIC SCHEDULING ALGORITHMS

A. PREVIOUS RESEARCH

Research previously done on the static scheduler is associated with the Computer Aided Prototyping System (CAPS) and the Prototype System Description Language (PSDL).

1. CAPS

CAPS is designed specifically as a development tool for hard real-time systems. Its primary objective is the computer-aided construction of software prototypes by retrieving connecting and adapting reusable software components from an online database via a formal prototyping language. It consists of three primary subsystems: a user interface, an execution support system, and a prototyping software base. Figure 4 on page 13 illustrates the three major components of CAPS.

The user interface consists of a syntax-directed editor for the formal prototyping language and a graphics tool for constructing and displaying data flow diagrams. The editor eliminates syntax errors by prompting the designers with appropriate alternatives at each step of the design process. The graphics tool provides a picture of the data flow diagrams which reinforces the text form of the system specifications [Ref. 29].

The execution support system consists of a translator which generates code to link reusable components together, a static scheduler which allocates time slots for prototype components prior to their execution, and a dynamic scheduler which allocates free time slots to non time-critical components as execution proceeds.

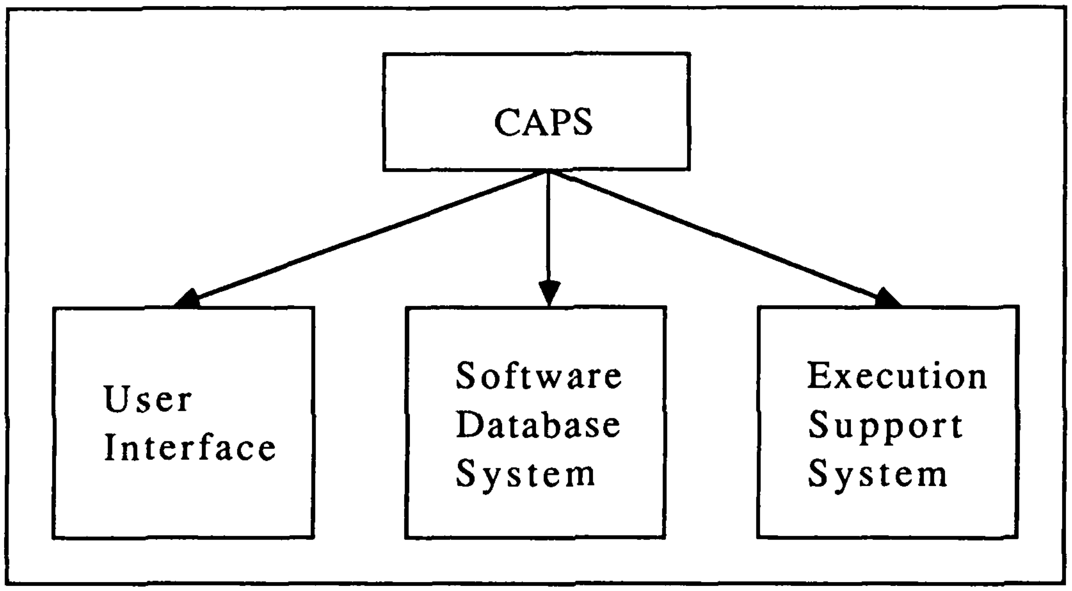


Figure 4 Major components of CAPS

The prototyping database consists of a design database, reusable software base, software design management system and a rewrite system. The prototyping database keeps track of designs and stores reusable prototype components together with their specifications.

The Software Design Management System (SDMS) is similar to a database management system with additional features required for computer-aided design applications. The SDMS is responsible for organizing, retrieving and instantiating the reusable software modules from the CAPS Database. The SDMS instantiates these modules as specified by the designer for execution of the current PSDL prototype. Overall, the SDMS supports efficient selection and retrieval of the relevant software modules [Ref. 28].

The CAPS Rewrite Subsystem provides a means for automatically generating uniform specifications for each reusable software module in the CAPS Software Database and for each PSDL lower level component. The Rewrite Subsystem, however, uses an approach which allows the designer to give more precise PSDL specifications. The Rewrite Subsystem then transforms each specification into a uniform or normal form to aid retrieval [Ref. 22]. Figure 5 on page 15 illustrates the architecture of CAPS.

2. PSDL

PSDL is a language designed for clarifying the requirements of complex real-time systems, and for determining the properties of the proposed designs for such systems via prototype execution. The language was designed to simplify the description of such systems and to support a prototyping method that relies on a novel decomposition criterion. PSDL is also the basis for a computer-aided prototyping system that speeds up the prototyping process by exploiting reusable software components and providing execution support for high level constructs appropriate for describing large real-time systems in terms of an appropriate set of abstractions [Ref. 22].

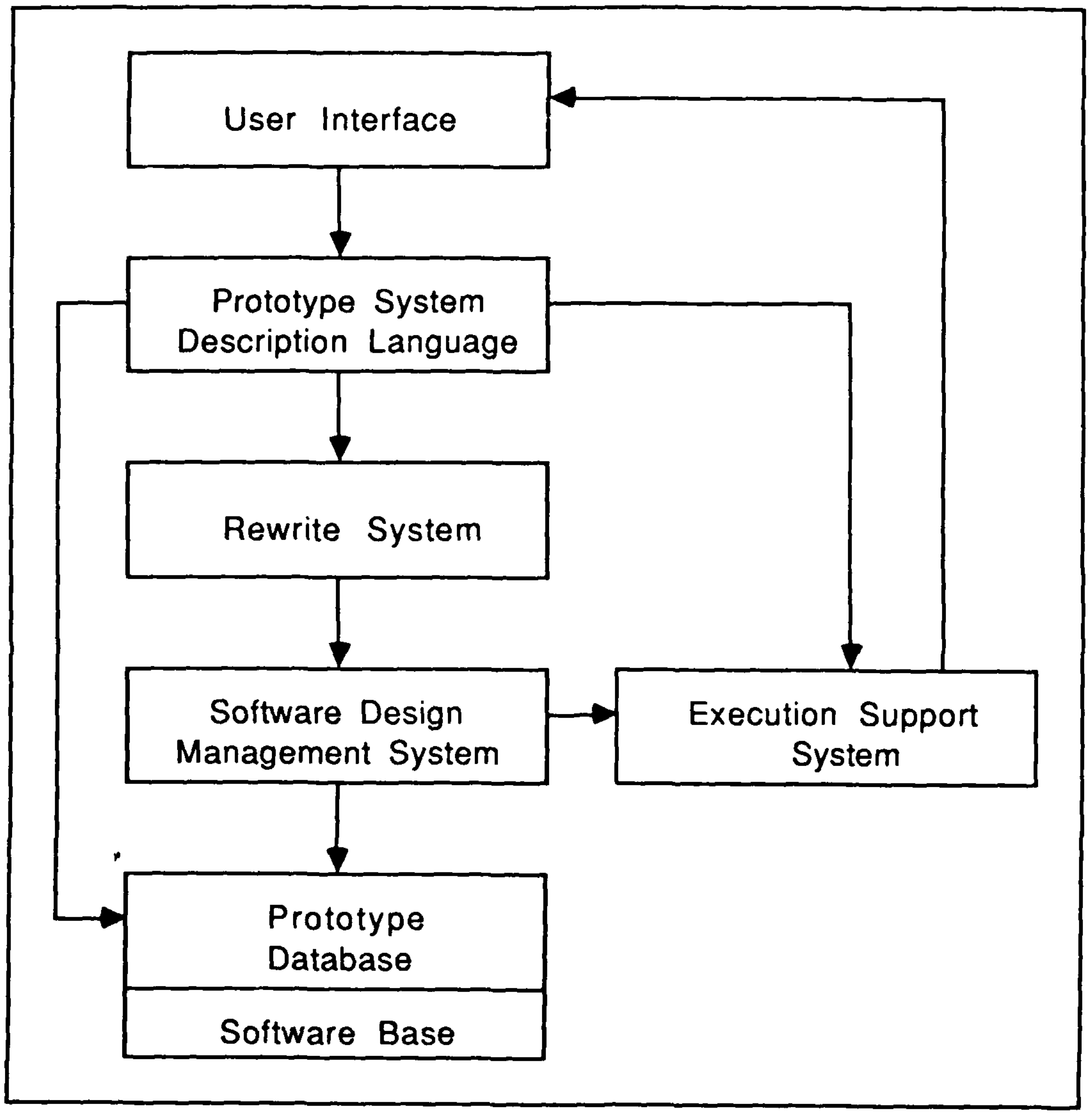


Figure 5 The Computer Aided Protyping System

PSDL supports the prototyping of large systems by providing a simple computational model that is close to the designer's view of real-time systems. The model is described in more detail below [Ref. 20].

a. Computational Model

To provide a small and portable set of PSDL constructs with a clear semantics, it is necessary to explore the mathematical model behind the language constructs. PSDL is based on a computational model containing operators that communicate via data streams.

Formally the computational model is an augmented graph

$$G = (V, E, T(v), C(v))$$

where V is the set of vertices, E is the set of edges, T(v) is the set of timing constraints for each vertex v, and C(v) is the set of control constraints for each vertex v. Each vertex is an operator and each edge is a data stream.

1. Operators: An operator is either a function or a state machine. When an operator fires, it reads one data object from each of its input streams, and writes at most one data object on each of its output streams.

Operators are either atomic or composite. Atomic operators cannot be decomposed in terms of the PSDL computational model. Composite operators have realizations as data and control flow networks of lower level operators.

2. Data Streams: A data stream is a communication link connecting exactly two operators, the producer and the consumer.

Each stream carries a sequence of data values. Streams have the pipeline property.

There are two types of data streams: data flow streams and sampled streams. A data flow stream guarantees that none of the data values is lost or replicated, while a sampled stream does not make such a guarantee. A data flow stream can be thought of as a fifo queue, while a sampled stream can be thought of as a cell capable of containing just one value, which is updated whenever the producer generates a new value.

b. Abstractions

Abstractions are an important means for controlling complexity. PSDL supports three kinds of abstractions: data abstractions, operator abstractions, and control abstractions. [Ref. 5]

1. Operator Abstractions: An operator abstraction is either a functional abstraction or a state machine abstraction. Both functional and state machine abstractions are supported by the PSDL constructs for operator abstractions. PSDL operators have two major parts: the specification and the implementation. The specification part contains attributes describing the form of the interface, the timing characteristics, and both formal and informal descriptions of the observable behavior of the operator. The implementation part determines whether the operator is atomic or composite.

2. Data Abstractions: All of the PDSL data types are immutable, so that there can be no implicit communication by means of side effects. Both mutable data types and global variables have been excluded from PSDL to help prevent coupling problems in large prototype systems.

3. Control Abstractions: The control abstractions of PSDL are represented as enhanced data flow diagrams augmented by a set of control constraints.

B. SURVEY OF STATIC SCHEDULING ALGORITHMS

This section includes a survey of Static Scheduling Algorithms for Hard Real-Time Systems. An overview of previous work and their characteristics are presented.

1. The Fixed Priorities Scheduling Algorithm

In many conventional hard real-time systems, tasks are assigned with fixed priorities to reflect critical deadlines, and tasks are executed in an order determined by the priorities. During the testing period, the priorities are (usually manually) adjusted until the system implementer is convinced that the system works. Such an approach can only work for relatively simple systems, because it is hard to determine a good priority assignment for a system with a large number of tasks by such a test-and-adjust method. Fixed priorities is a type of static scheduling. Once the priorities are fixed on a system it is very hard and expensive to modify the priority assignment.

2. The Harmonic Block with Precedence Constraints Scheduling Algorithm

This scheduling algorithm is being used by the CAPS [Ref. 34]. The data flow diagram is given in Figure 6 [Ref. 34] on page 19. The first component of the DFD, "Read_PSDL", reads and processes the PSDL prototype program. The output of this step is a file containing operator identifiers, timing information and link statements.

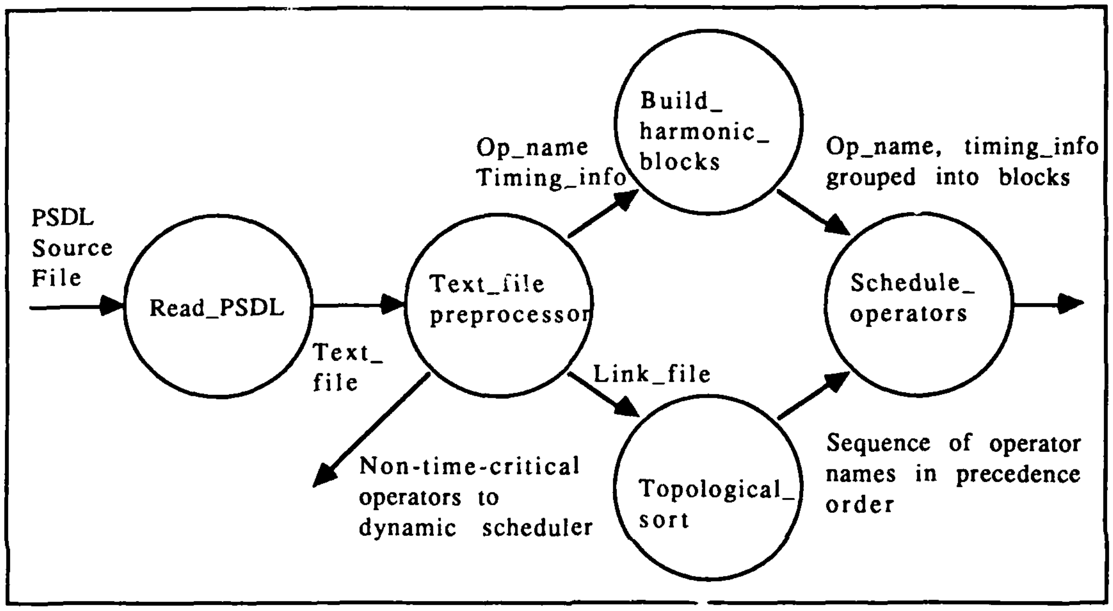


Figure 6 First level data flow diagram

The second component is "Pre-Process_File". The file generated in the first step is analyzed and the data is divided into three separate files based on its destination or additional processing required. The Non-Crits contains the data of all noncritical operators. The Operator file contains all critical operator identifiers and their associated timing constraints. The Links file contains the link statements which syntactically describe the PSDL implementation graphs. During this step some basic validity checks on the timing constraints are performed. If any of the checks fails an exception is raised and an appropriate error message is submitted to the user.

The "Sort_Topological" component performs a topological sort of the link statements contained in the Links file. The requirements for a topological sort imply that the statements being sorted do not have circular dependences. These properties define the execution precedence of the time critical operators. The output is a precedence list of critical operators stipulating the exact order in which they must be executed.

The second output of "Pre-Process_File", the Operator file, is the input to "Build_Harmonic_Blocks". A harmonic block is defined as a set of periodic operators where the periods of all its component operators are exact multiples of a calculated base period. All the operators must be periodic. The sporadic operators are converted to their periodic equivalents. The periodicity helps insure that execution is completed between the beginning of a period and its deadline.

In order to convert a sporadic operator into its equivalent periodic operator the following parameters of the sporadic operator must be known

Maximum Execution Time (MET).

Minimum Calling Period (MCP).

Maximum Response Time (MRT).

Some rule< must be obeyed by the parameters described above to obtain an equivalent periodic operator, the rules are the following:

MET < MRT. This rules insures that (MRT - MET) produces a positive value.

MET < MCP. This restriction insures that the period calculated will conform to a single processor environment.

The periodic equivalent is then calculated as P = min (MCP, MRT - MET). The value of P must be greater than MET in order for the operator to complete execution within the calculated period on a single processor.

After all the operators are in periodic form, they are sorted in ascending order based on the period values. A second preliminary step is to calculate the base block and its period for the sorted sequence of operators. The base period is defined as the greatest common divisor (GCD) of all the operators in one sequence that will be scheduled together.

The last preliminary step is to evaluate the length of time for the harmonic block. The actual harmonic block length is the least common multiple (LCM) of all the operators' period contained in the block. The harmonic block and its length are an integral part of the static schedule. This block represents an empty time frame within which the operators will be allocated time slots for execution.

The outputs of "Sort_Topological" and "Build_Harmonic_Blocks" are used by "Schedule_Operators" in order to create a static schedule. The static schedule is a linear table giving the exact execution start time for each critical operator and the reserved MET within which each operator completes its execution.

This linear table is constructed in two iterative steps. In the first step operators are considered in the order determined by the topological sort and an execution time interval is allocated for each operator based on the equation INTERVAL = [current time, current time + MET]. Next the process creates a firing interval for each operator during which the second iterative step must schedule the operator. The firing interval stipulates the lower and upper bound for the next possible start time for an operator based on its period. The second step, uses the lower bound of each firing interval when it schedules operators during subsequent iterations. The sequence of operators is allocated time slots according to the earliest lower bound first. Before an operator is allocated a time slot, this step verifies that:

(current time + MET) <= harmonic block length and current time <= upper bound of firing interval.

This condition is applicable to every operator scheduled in that harmonic black. This step also calculates new firing intervals for each operator scheduled. Once all the operators are correctly scheduled within an entire harmonic block a static schedule is available. All subsequent harmonic blocks are copies of the first.

A theoretical development and implementation guideline of this algorithm is available in the [Ref. 34] and [Ref. 19].

The actual implementation of this algorithm and the analysis of its performance is described in the [Ref. 30].

3. The Earliest Start Scheduling Algorithm

This algorithm considers the scheduling of n tasks on a single processor. Each task becomes available for processing at time a_i, must be completed by time b_i, and requires d_i time units for processing.

There are two versions of the criteria [Ref. 8]: one allows job splitting (preemptable tasks), under this assumption it is only required to complete d_{ki} units of processing between a_i and b_i (where $d_{1i} + d_{2i} + ... + d_{mi} = d_i$, and m is the total number of splits of the task i); and the other version assumes that job splitting is not allowed (nonpreemptable tasks). [Ref. 8]

a. Preemptable Version

Consider the rectangular matrix that has a column for each job and a row for each unit of time available. In this matrix it is necessary to distinguish between admissible and inadmissible cells. For job i the cell (i, j) is admissible if $a_i < j <= b_i$ and inadmissible otherwise. The admissible cells correspond to the times where the task may be performed.

Associated with each row is an availability of one unit of time, and with each column a requirement of d_i units of time. If the task i is being processed at time j, a 1 is placed in the admissible cell. This problem is equivalent to that of finding a set of 1's placed in admissible cells such that columns sums satisfy the requirements di and each line contains at most one single 1.

The preemptable earliest start scheduling algorithm does not account for precedence constraints. In order to include the precedence constraints in this algorithm it is necessary to do some modifications. The modification can utilize some concept like the harmonic block discussed in the former algorithm and also include the constraints that a job j that is preceded by a job i is admissible only after i is scheduled.

[Ref. 3] presents an implementation in FORTRAN to solve the case without precedence constraints. This type of algorithm does not account for precedence constraints, and is not applicable to our case because it assumes that all the tasks are preemptable.

This algorithm is bounded by O(n) in time, and as most heuristic algorithms, does not guarantee that the solution (assuming that at least one is available for the problem) is found.

b. Nonpreemptable Version

In this approach, also, the precedence constraints are not included in the analysis, but they may be easily taken into account during the construction of all the feasible sequences.

The main idea is to enumerate implicitly all the possible orderings by a branch, exclude and bound algorithm. During the branch all infeasible sequences due to violation of the due date are discarded (here is possible to include the precedence constraints).

All the possible sequences are enumerated by a tree type construction. From the initial node we branch to n new nodes on the first level of descendent nodes. Each of these nodes represents the assignment of task j, 1 <= j <= n, to be the first in the sequence. Associated with such a node there is the completion time t_{ij}, of the task j in the position i, i.e., $t_{1i} = a_i + d_i$. Next we branch from each node on the first level to (n-1) nodes on the second level. Each of these nodes represents the assignment of each of the (n-1) unassigned tasks to be second in the sequence. As before, we associate with the corresponding node the completion time of the task $t_{2j} = \max (t_{1i}, a_j) + d_j$. We continue in similar fashion. The initial node is a dummy node. In the unconstrained case all the nodes must be present in the level 1 (level 0 is assumed to be the dummy root of the complete tree). In case with precedence constraints, we allocate only the tasks that have only external input or no predecessor in the level 1.

Consider the (n-k+1) new nodes generated at the level k of the tree construction. If the finish time t_{kj} associated with at least one of these nodes exceeds its due date then the subtree rooted at the infeasible node may be excluded from further consideration.

The bounding condition applies only when seeking an optimal ordering of the sequence that minimizes the length of the block.

A more detailed explanation, as well a step by step definition of the algorithm, may be found in [Ref. 3: p. 514-519].

Another possible implementation of this algorithm is to utilize the concepts, length of the harmonic building block and the firing interval for each task, described in the former algorithm. In order to include the precedence constraints and the period of the operators the following variant scheme was developed [Ref. 34]:

define the agenda list as an empty list, define the waiting set as an empty set,

define the successors and predecessors of each task,

evaluate the length of the harmonic block,

start loop,

select the tasks that have no predecessors and put them in the waiting set,

select the task from the waiting set that has the smallest earliest start time (if ties occur, then some other criteria must be applied) and put this task in the agenda list as the last component; if the waiting list is empty then stop (the agenda list contains the schedule), otherwise repeat the loop.

The algorithm described above is not optimal, but has the advantage that it is more compact in time and space. The main deviation of the algorithm described above from the idea expressed in [Ref. 8] is that this algorithm does not take into account all the possible branches. Once a decision about more than one branch is made, it is not possible to come back and test the other branches later. Another possible version of this algorithm is to consider the earliest deadline instead the earliest start time, as criteria to include a task in the agenda list. The description of the algorithm needs to be slightly modified. An implementation of the variant described above is available in [Ref. 20].

4. The Minimize Maximum Tardiness with Earliest Start

This algorithm considers a sequencing problem consisting of n tasks and a single processor. Task i is described by the following parameters:

1. The ready time (a_i), the earliest point in time at which processing may begin on i (i.e., an earliest start time).

2. The processing time (d_i), the interval over which task i will occupy the processor.

3. The due date (b_i), the completion deadline for task i.

These three parameters a_i, b_i, and d_i are known in advance and no preemption is allowed in the processing of the tasks.

Suppose task i is completed at time C_i. Then task i will be tardy if $C_i > b_i$. The tardiness of task i, T_i is defined to be max $\{0, C_i - b_i\}$. The scheduling objective is to minimize the maximum task tardiness, T_{max}, which is simply equal to max $\{T_i\}$.

For the static version of the n tasks single processor problem without precedence constraints (all a_i's equal), Tmax is minimized by the sequence $b_1 <= b_2 <=...<= b_n$, that is, by processing the tasks in nondecreasing order of their deadlines. [Ref. 6: p. 172]

In the dynamic version of the problem the statement above can also be applied if the tasks can be processed in a preemptable fashion, in this case sequencing decisions must be considered both at task completion and at task ready time as follows:

At each task completion, the task with minimum b_i among available tasks is selected to begin processing.

At each ready time, a_i, the deadline of the newly available task is compared to the deadline of the task being processed. If b_i is lower, task i preempts the task being processed, otherwise the task i is simply added to the list of available tasks.

The solution to the preemptive case is not difficult to construct because the mechanism is a dispatching procedure. Since all nonpreemptive schedules are contained in the set of all preemptive schedules, the optimal value of Tmax in the preemptive case is at least a lower bound on the optimal Tmax for the nonpreemptive schedules. This principle is the basis for the algorithm.

In the nonpreemptive problem, there is a sequence corresponding to each permutation of the integers 1, 2, ..., n. Thus there are at most n! sequences, but many of these sequences do not need to be considered. The number of feasible sequences depends on the data in a given problem, but will usually be much less than n!.

Based on this observation, a "branch and bound" algorithm has been used to systematically enumerate all the feasible permutations[Ref. 6]. (Although the authors in [Ref. 6] referred to their approach as branch and bound, their algorithm actually finds an optimal schedule using an A*-search strategy.)

The branching tree is essentially a tree of partial sequences. Each node in the tree at level k corresponds to a partial permutation containing k tasks. Associated with each node is a lower bound on the value of the maximum tardiness which could be achieved in any completion of the corresponding partial sequence (obtained using the preemptive adaptation). The calculation of a lower bound allows the algorithm to eliminate many inferior sequences ahead of time. If the bound associated with some partial sequence is greater than or equal to Tmax, the best maximum tardiness of the complete sequences found so far, then it is not necessary to complete the partial sequence in the search for optimum solution.

The "branch and bound" algorithm maintains a list of nodes ranked in nondecreasing order of their lower bounds. At each stage the node at the top of the list is removed and a set of new nodes corresponding to the augmented partial sequences is added to the list.. These nodes are formed by appending one unscheduled task to the removed partial sequence. The algorithm terminates when the node at the top of the list corresponds to a complete sequence. At this point, the complete sequence attains a value of Tmax which is less than or equal to the lower bound associated with every partial sequence remaining on the list, and the complete sequence is therefore optimal.

Before the tree search begins, the algorithm uses a heuristic initial phase to obtain a good initial solution to the problem. There is no guarantee that the initial solution meets all of the deadlines. This initial solution allows the tree search to begin with a complete schedule already on hand, and allows several partial schedules to be discarded in the course of the tree search, simply because their lower bound exceeds the value of the initial solution.

There are four heuristics available:

Ready time: sequence the tasks in nondecreasing order of their ready time, a_i.

Deadline: sequence the tasks in nondecreasing order of their deadlines, b_i.

Midpoint: sequence the tasks in nondecreasing order of the midpoints of their ready times and deadlines $(a_i + b_i)/2$, i.e use the nondecreasing order of $a_i + b_i$.

PIO: sequence the tasks in the order of their first appearance in the optimal preemptive schedule, which is constructed by the dynamic version.

[Ref. 6: p. 171-176] contains a complete and detailed description of the algorithm. This algorithm does not take into account the possible precedence constraints among the tasks, but these precedence constraints can be taken into account during the evaluation of the branch and bound solution of the tree search. The inclusion of the precedence constraints in the evaluation of the heuristics must also be considered. The algorithm can be extended to handle the case where tasks can only be started after some time in the future (this happens when some of the tasks are periodic), the modification needed is in the definition of task's scheduled start time.

5. The Deadline and Criticalness Scheduling Algorithm

This algorithm is based upon the following assumptions:

1. All application tasks are known, but their invocation order is not known. That is, tasks arrive dynamically and independently.

2. There are no precedence constraints on the tasks; they can run in any order relative to each other as long as deadlines are met.

3. Each task has the following characteristics: an arrival time (a_i) that is the time at which the task is invoked; a worst-case computation time (d_i) that is the maximum time needed for it completion; a criticalness (n_i) that is one of the n possible levels of importance of the task; a deadline (b_i) that is the time by which the task has to complete execution. These characteristics are time invariant.

The algorithm discussed in this subsection assumes the existence of an environment that consists of a distributed system consisting of N nodes. Each node contains m processors divided into two types: systems processors dedicated to executing system tasks and application processors executing only application tasks The connection medium for the nodes is assumed to be a shared bus. In other words the system under analysis consists of a collection of multi-processors connected together in a loosely-coupled network.

The main systems of interest to the discussion are the local scheduler and the global scheduler. The local scheduler at each node maintains a data structure called the System Task Table (STT); this table contains a list of applicable tasks that have been dynamically guaranteed to make their deadline at this local node. Entries in the STT are arranged in the order of execution and tasks are dispatched for execution from this table. Each STT entry, corresponding to a guaranteed task, has five attributes: the arrival time, the latest start time, the criticalness, the deadline, and the computation time.

The Local Scheduler, which can re-order, insert or remove any entries in the STT, is activated upon the arrival of a new task at the local node, or in response to a bidding process which is initiated by the global scheduler. The Local Scheduler, working with a copy of the STT, determines if a new task can be inserted into the current STT such that all previous tasks in the STT as well as the new task meet their deadlines. If so, then the task is guaranteed and the latest start time is determined. If the new task cannot be guaranteed locally, or can only be accommodated at the expense of some previously guaranteed task(s), then the rejected task(s) is(are) handed over to the Global Scheduler.

The Global Scheduler then takes the necessary actions to transfer the task(s) to any alternative nodes that may have the resources to accept this(those) task(s). The Global Scheduler uses bidding. Request-for-bids (RFB) are broadcast to the other nodes when a local task has to be reallocated. If several remote nodes respond with bids reflecting their surplus, the Global Scheduler evaluates those bids and transfers the task to the node with the best bid.

The algorithm first attempts to guarantee an incoming task according to its deadline, ignoring its criticalness. If the task is guaranteed then the scheduling is successful. However, if this first attempt at scheduling fails, then there is an attempt to guarantee the new task at the expense of previously guaranteed, but less critical tasks. If enough less critical tasks can be found then the new task is guaranteed at this site and the removed tasks are transferred to alternative sites. If there are not enough less critical tasks, or the deadline of the new task is such that the removal of any such tasks does not allow the new task to meet its deadline, then the new task is transferred to an alternative site. The process is repeated at the next node until the task either meets its deadline or its deadline expires.

A detailed explanation of the algorithm above, discussing all the steps as well the performance is contained in [Ref. 7: p. 152-160].

6. The Optimal Static Scheduling Algorithm

The optimal static scheduling algorithm using enumeration techniques is guaranteed to find a feasible schedule if one exists and improves on the current version of the algorithms being utilized by the CAPS system for this reason. It is a slow algorithm, but also a reliable algorithm, in the sense that the structures and concepts utilized are very simple. A detailed explanation is in [Ref. 8].

III. DESIGN OF A BRANCH AND BOUND STATIC SCHEDULING ALGORITHM

In this Chapter we present an efficient approach for the optimal scheduling for a single processor. The main point of this approach is to use the branch and bound method to save search time and memory space. Before we present this approach we introduce some definitions of terms and functions.

A. PRELIMINARY DEFINITION

MET(i): task i requires MET(i) time units of processing,

PERIOD (i): period of the base operator for the task i,

PHASE(i): phase of the base operator for the task i,

INSTANCE(i): instance number of the task i,

OPERATOR(i): operator ID of the task i;

EARLIEST_START(i): earliest start possible for the task i,

TIME_ALLOWED(i): maximum time allowed to finish the task i after the earliest start,

DEADLINE(i): maximum completion time allowed for the task i,

TARDINESS(i): the amount of time by which i missed its deadline,

COMPLETION(i): time when the task i is finished.

Figure 7 [Ref. 29] on page 35 illustrates the timing constraints for a periodic operator in PSDL.

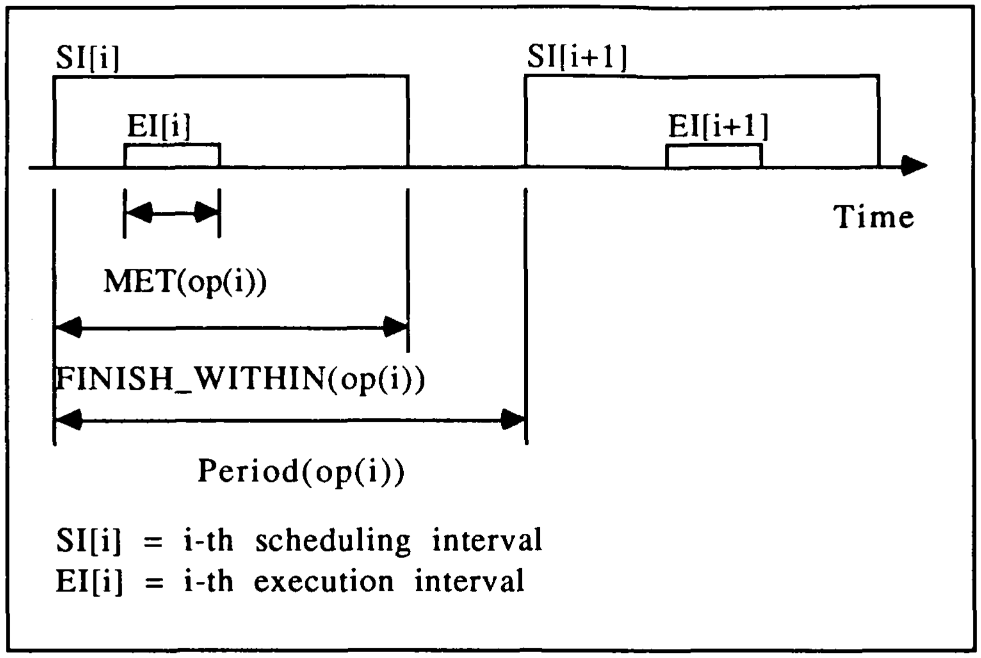


Figure 7 Timing constraints for a periodic task

B. GENERATE THE GRAPH OF CONSTRAINTS

We use the following steps to obtain the graph of constraints.

1. obtain the GCD for the all operators' periods,

```
GCD(i,j) := if j > i then GCD(j,i)
                 else if i mod j = 0 then return j;
               else GCD(j, (i mod j));
```

2. obtain the LCM for the all operators' periods,

```
LCM(i,j) := i * j / GCD(i, j);
   Length_of_harmonic_block := LCM(periods);
```

3. obtain the number of tasks of each operator,

```
   number of tasks(i) := length of harmonic block/period(i);
```

4. generator the chains of tasks,

```
    precedence(i,j) = '1'
  if OPERATOR(i) = OPERATOR(j) and INSTANCE(i) < INSTANCE(j)
```

5. generator interconnect chains.

```
    precedence(i, j) = '1'
  if OPERATOR(i) /= OPERATOR(j) and
    PERIOD(i) * INSTANCE(i) = PERIOD(j) *INSTANCE(j);
```

Figure 8 on page 37 illustrates the first level DFD graph of constraints.

C. SCHEDULED LIST COST

The cost of an indexed sequence of scheduled tasks is the maximum tardiness value of the task in the list.

1. Algorithm for List Cost

```
tardiness : integer;
list_cost : integer := minus infinity;
for i in 1 .. task_length loop
    tardiness := get_tardiness(i);
    if list_cost < tardiness then
        list_cost := tardiness;
    end if;
end loop;
```

We can get the tardiness value by using the following equations:

Completion(i) = Start(i) + MET(op(i));

Completion(0) = 0;

Earliest_Start(i) = Instance(i) * Period(i) + Phase(i);

Phase(op(i)) = Earliest_Start(j) where (op(j)) = (op(i)) and Instance(j) = 0;

Deadline(i) = min(Earliest_Start(i) + Finish_Within(op(i)) Length_of_harmonic_block);

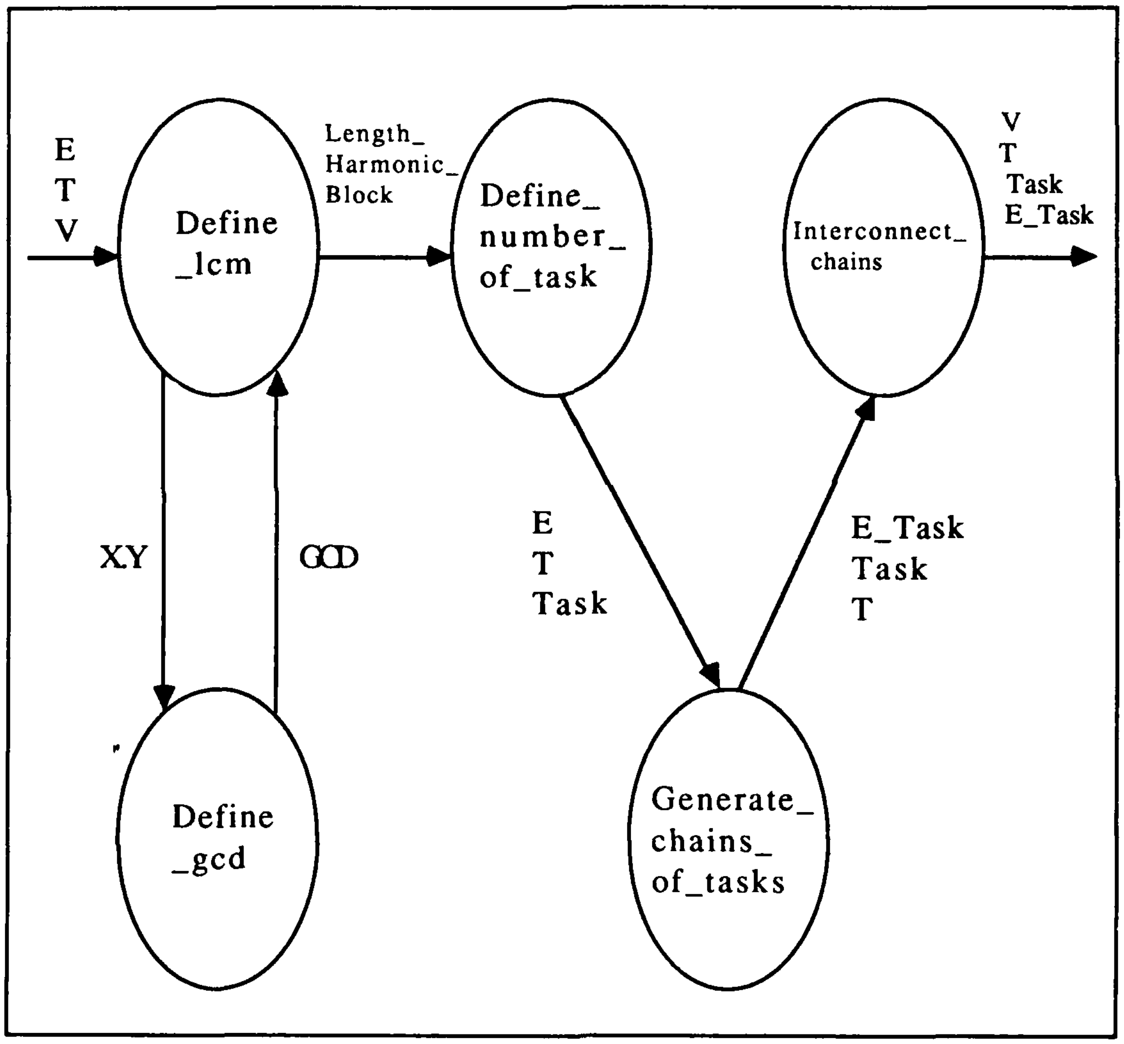


Figure 8 The first level DFD graph of constraints

Phase(k) is defined as zero, where k is the first task of the scheduling sequence.

D. LOWER BOUND ON THE COST OF THE UNSCHEDULED TASKS

In the alogrithm, a lower bound on the cost of the all unscheduled tasks is needed for the branch and bound decissions. To obtain such a lower bound, we first choose a lower bound for the Estimate_completion of each unscheduled task and choose an upper bound for the Estimate_deadline of each unscheduled task. Then we define the estimate cost of each unscheduled task as the diffence of its corresponding Estimate_completion and Estimate_deadline. In order to obtain the lower bound of the Estimate_completion, we assume that all unscheduled tasks are executed one after another and that there is no idle time interval between them. So we choose the summation of MET(i) as the executing time where i is the unscheduled tasks.

We define estimate_value as follows:

estimate_value := max(estimate_cost(i)) where i represents the tasks which are unscheduled.

estimate_cost := estimate_completion - estimate_deadline;

sum(k) := summation MET(i);

where i is the task which is the ancestor of task(k) and unscheduled.

estimate_completion(k) := completion(n) + sum(k);

where n is the last task of the scheduled list.

estimate_deadline(k) := period(k) * instance(k) + estimate_phase(k) + finish_within(k);

```
completion(schedule) := completion(last task of schedule);
start(n) := max(completion(schedule), earliest_start(n));
earliest_start(n) := phase(n) + period(n) * instance(n);
estimate_phase(n) := start(k) if there is a task k included in
    the scheduled list where operator(k) = operator(n) and
    instance(k) = 0, and estimate_phase(n) = period(n) otherwise,
  since 0 <= phase <= period.
```

1. Algorithm for Get_estimate_value

```
begin
    estimate_cost := minus infinity;
    for k in nodes(g) loop
        -- g contains only the unscheduled tasks
          sum := 0;
          for each m in g such that m in ancestors(k) loop
                sum := sum + MET(m);
          end loop;
          estimate_cost := completion(n) + sum - deadline(k);
          if estimate_value < estimate_cost then
                estimate_value := estimate_cost;
          end if;
    end loop;
end;
```

E. BRANCH AND BOUND STATIC SCHEDULING ALGORITHM

This algorithm is used to find the feasible or optimal scheduling by branch and bound techniques.

1. The Algorithm for Branch and Bound Static Scheduling

```
 find_schedule (g : graph; schedule: out sequence{task};
                    feasible : out boolean) is
      -- g represents the precedence constraints of the unscheduled
     --tasks
best_cost : integer := infinity;
begin
      branch_and_bound(g, [ ], best_cost, schedule);
     feasible := (best_cost <= 0);
end  find_schedule;


branch_and_bound(g : graph; s : sequence{task};
                         best_cost : in out integer;
                          best_schedule : out sequence{task}) is
begin
      if g is empty and then cost (s) < best_cost
      then best_cost := cost(s);
            best_schedule := s;
    end if;
      for each node n in g such that predecessors (n) = { } loop
         if max (cost (s || [n]), least_cost (g - n, s || [n])) < best_cost
            then branch_and_bound (g - n, s || [n], best_cost,
                                           best_schedule);
           if best_cost <= 0 then return;
```

```
            end if;
        end loop;
    end  branch_and_bound;


    least_cost (g, s) = max k: node in g of
                {completion_time_of_last_task_in_s - deadline(k)
                  + sum (met (m) such that m in ancestors(k) and (not m in
          s))}
```

2. Example

We will give an example to illustrate this algorithm. Suppose that we have four operators with precedence relationships show in Figure 9. The timing constraints are described as follows:

operator	period	met	finish_within	predecessors
1	15	2	10	{ }
2	15	1	10	{ 1 }
3	30	3	10	{ 1 }
4	30	2	10	{ 2,3 }

By using the method of finding the harmonic block described in [Ref. 34],

LCM(periods) = 30,

number_of_tasks(1) = 30/15 = 2,

number_of_tasks(2) = 30/15 = 2,

number_of_tasks(3) = 30/30 = 1,

number_of_tasks(4) = 30/30 = 1.

total_number_of_tasks = 2 + 2 + 1 + 1 = 6;

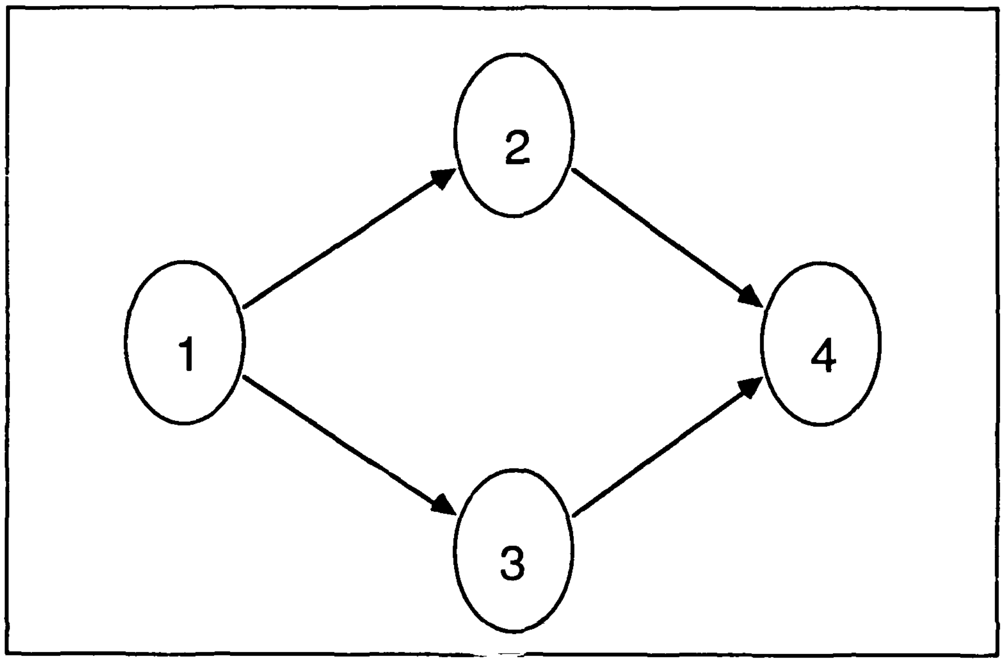


Figure 9 Precedence graph of operators

A task is an instance of an operator in the scheduling interval. We can get the constraint graph of tasks as follows:

operator(task(1)) = 1;	instance(task(1)) = 0;
operator(task(2)) =1;	instance(task(2)) = 1;
operator(task(3)) = 2;	instance(task(3)) = 0;
operator(task(4)) = 2;	instance(task(4)) = 1;
operator(task(5)) = 3;	instance(task(5)) = 0;
operator(task(6)) = 4;	instance(task(6)) = 0;

Figure 10 on page 43 illustrates the constraint graph. According to the graph of constraints we can get the ancestors of any task:

ancestors(1) = 1;	ancestors(2) = 1, 2;
ancestors(3) = 1, 3;	ancestors(4) = 2, 3, 4;
ancestors(5) = 1, 5;	ancestors(6) = 3, 5, 6;

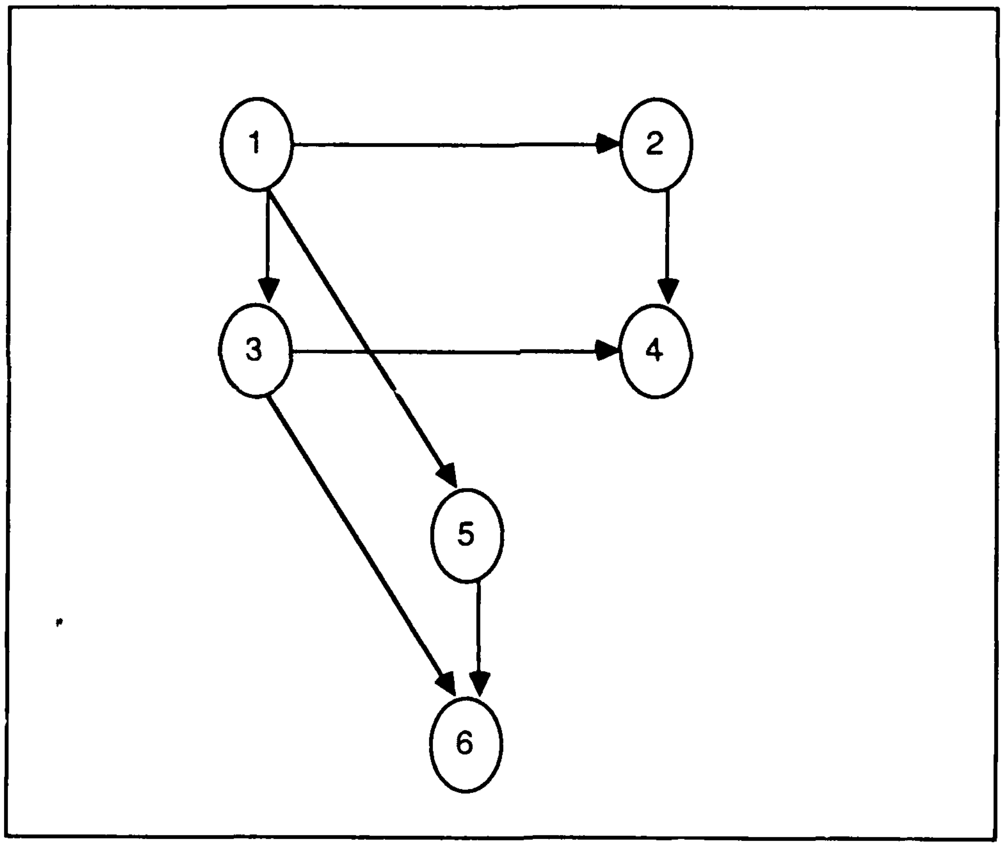


Figure 10 Constraints graph of tasks

We expand the graph in Figure 10 to a search tree, illustrated in Figure 11 on page 44. Although the search tree may be very large, it is not necessary for every path to be open. We show the

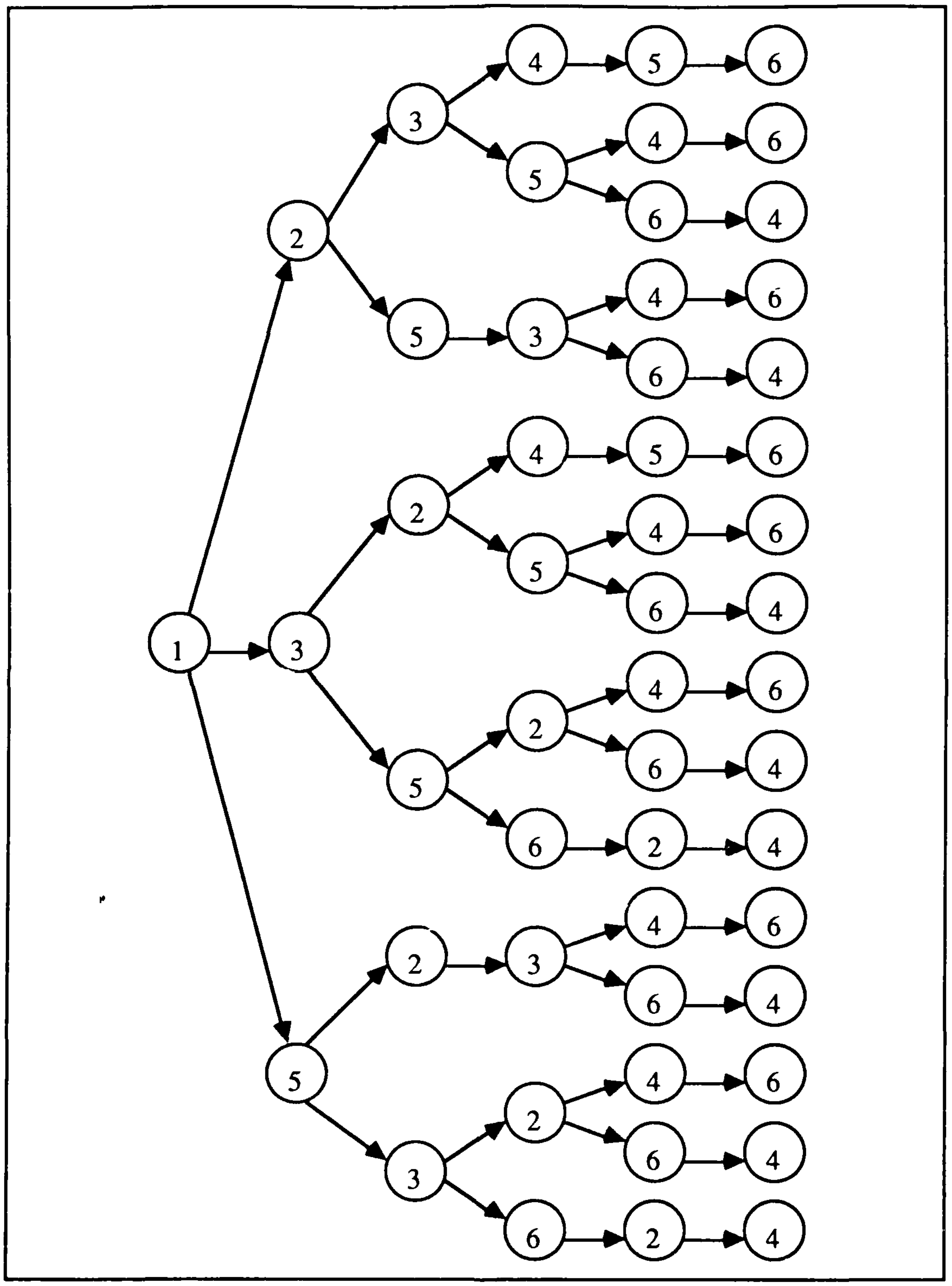


Figure 11 Search graph of the schedule

calculation of the lower bound for the case in which the schedule s = [1, 2, 3].

list_cost = max(tardiness(x)); x = 1, 2, 3;

tardiness(1) = completion(1) - deadline(1)

= (0 + MET(1)) - (0 + finish_within(1))

= (0 + 2) - (0 + 10)

= -8

tardiness(2) = completion(2) - deadline(2)

= (max(completion(1), phase(2) + instance(2) * period(2)) + MET(2)) - (max(completion(1), phase(2) + instance(2) * period(2)) + period(2))

= (max(2, 15) + 2) - (max(2,15) + 10)

= -8

tardiness(3) = (max(17, 17) + 1) - (max(17, 17) + 10)

= -9

list_cost = max(-8, -8, -9) = -8

least_cost = max(estimate_cost(x)); x = 4, 5, 6

estimate_cost(5) = estimate_completion(5) - estimate_deadline(5);

estimate_completion(5) = max(completion(3), phase(op(5)) + period(op(5)) * instance(5)) + MET(op(5));

estimate_deadline(5) = min(max(completion(3), phase(op(5)) + period(op(5)) * instance(5)), length_of_harmonic_block);

completion(3) = max(completion(2), phase(op(3))) +

(period(op(3)) * instance(3)) + MET(op(3)));

completion(3) = 17 + 15 * 0 + 1 = 18;

phase(op(5)) = 30;

phase(op(5)) + period(op(5)) * instance(5) = 30 + 30 * 0 =30;

estimate_completion(5) = max(18, 30) + 3 =33;

estimate_deadline(5) = min(max(18, 30), 30) = 30,

estimate_cost(5) = 33 - 30 = 3;

using the same rules we can get

estimate_cost(4) = -3

estimate_cost(6) = -5

least_cost = max(-3, 3, -5) = 3,

max_value = max(list_cost, least_cost)

= max(-8, 3) = 3

In this case max_value < best_cost, which equal to infinity initially, so the subtree rooted at 4 cannot be pruned, the partial schedule becomes list = [1, 2, 3, 4]; and the search continues.

At the end of the first leaf of the search tree, we get :

list = [1, 2, 3, 4, 5, 6];

Since we set the best_cost = infinity at the beginning so

list_cost < best_cost

best_cost := 8

best_sequence = [1, 2, 3, 4, 5, 6]

.

.

At the end of the second leaf of the search tree, we get:

best_cost := 5

best_sequence =[1, 2, 3, 5, 4, 6];

.

.

At the end of the third leaf of the search tree, we get:

best_cost := 3;

best_sequence = [1, 2, 3, 5, 6, 4];

.

.

When the searching process reach the fourth path:

list = [1, 2, 5, 3];

list_cost := 6;

least_cost := 8;

max_value := 8;

best_cost := 3;

Since max_value > best_cost so we do not need to consider any extensions of the partial sequence [1, 2, 5, 3], and the search continues with the partial sequence [1, 3].

.

.

Finally we get the answer:

best_cost := -7;

best_sequence = [1, 3, 2, 4, 5, 6];

F. SUMMARY OF THE BRANCH AND BOUND SCHEDULING ALGORITHM

This algorithm guarantees that it can find a feasible schedule if one exists, or if no feasible schedule exists, then a schedule with the minimum possible tardiness.

IV. IMPLEMENTATION OF STATIC SCHEDULERS

In this chapter we implement two optimal static scheduling algorithms: the exhaustive enumeration and the branch and bound algorithm. The Ada programming language has been used as the basic implementation language of the static scheduling algorithms in this thesis.

A. GENERAL CONCEPT OF THE EXHAUSTIVE ENUMERATION SCHEDULING ALGORITHM

This algorithm includes two steps to obtain the goal which is feasible for the static scheduler. The first step is to generate the graph of constraints. The second step is to get the optimal scheduling by enumeration techniques.

1. Generate the Graph of Constraints

The graph of constraints is completely defined and evaluated by using the following steps:

1. Evaluation of the GCD of the operators.
2. Evaluation of the LCM of the operators.
3. Evaluation of the number of tasks in the graph of constraints.
4. Generation of chains of tasks.
5. Interconnection of the chains.

From steps 1 and 2, we can get the length of harmonic block. The length of the harmonic block is simply the least common multiple (LCM) of all the operators that belongs to the set being analyzed. Z is defined as the LCM of (X,Y) if and only if Z mod X = 0 and Z mod Y = 0 and (W mod X = 0 and W mod Y = 0) implies that Z <= W. The LCM is computed by taking two periods at a time, multiplying them together, and then dividing this result by the greatest common divisor (GCD) of the two periods. This result is then multiplied together with the next period and divided by their GCD until all operators in the set have been processed. The result of this operation on the last pair in the set is the LCM of all operators in the set.

Step 3 finds the number of tasks for each operator, obtained by dividing the length of the harmonic block by the period of the operator. By the precedence constraints of the operators, we can get the precedence constraints of the tasks and generate the graph of constraints from steps 4 and 5.

The enumeration techniques require that if i is predecessor of j (E_TASK(i,j) = 1), then the integers associated with them must obey the relation n(i) < n(j). To ensure that the graph of constraints obeys this relation we use step 6 to reorder the tasks.

2. Optimal Scheduling by Enumeration Techniques

There are two approaches to obtain the feasible (optimal) sequences. One approach is by explicit enumeration and the other is implicit enumeration.

a. Explicit Enumeration

The steps necessary to obtain the optimal enumeration scheduling algorithm, in this approach, are:

1. Obtain the ancestors and descendants of each task,

2. Obtain the maximum lexicographical order legal sequence,

3. Generate all the possible legal sequences,

4. Apply the cost function to each legal sequence generated, until one of them is feasible (optimal).

Figure 12 [Ref. 8] on page 51 illustrates the first level DFD for explicit enumeration.

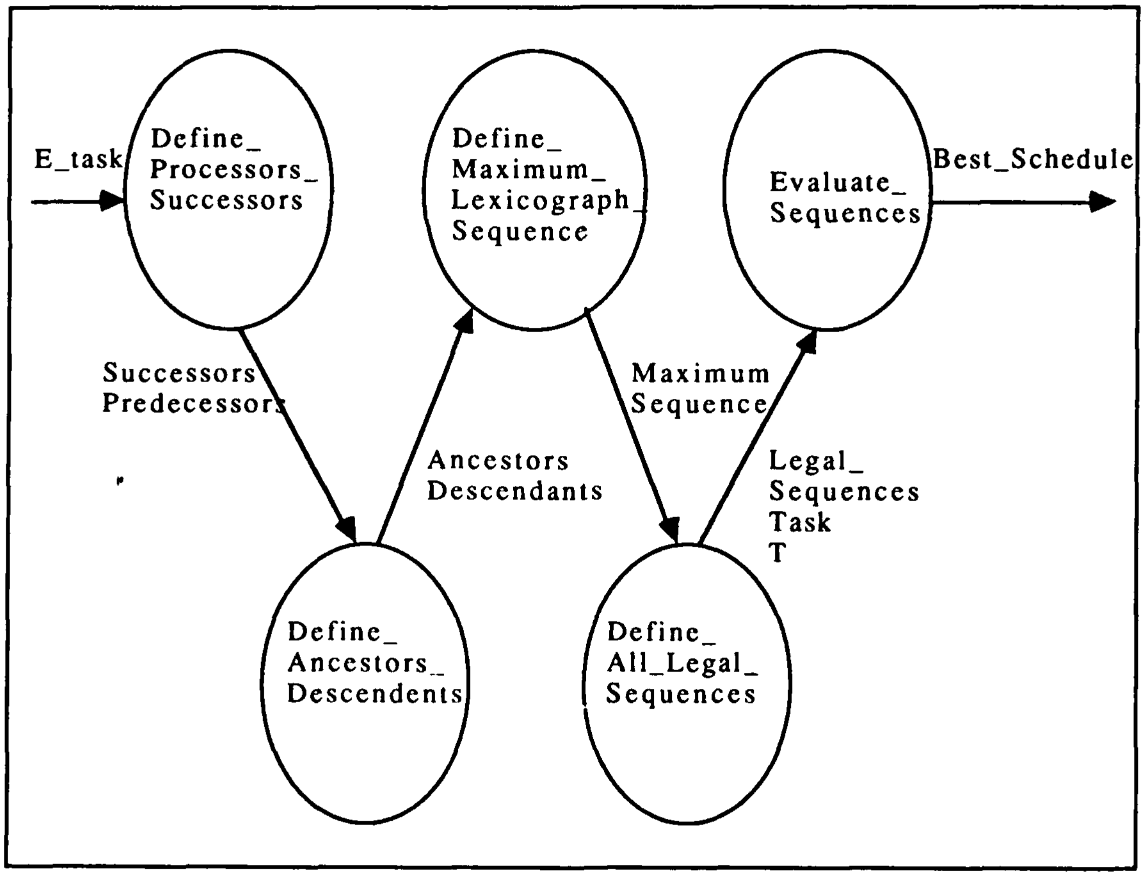

E_task
Define_ Processors_ Successors
Define_ Maximum_ Lexicograph_ Sequence
Evaluate_ Sequences
Best_Schedule
Successors Predecessors
Ancestors Descendants
Maximum Sequence
Legal_ Sequences Task T
Define_ Ancestors_ Descendents
Define_ All_Legal_ Sequences

Figure 12 The first level DFD for Explicit Enumeration

The generation of the ancestors and descendants of each task is constructed recursively from the predecessors and successors of each task.

To evaluate the ancestors of the task i, we begin by including in the set ANCESSORS(i) all the predecessors of the task i After this, we include in this list all the ancestors of all the predecessors of the task i. The construction of the DESCENDANTS of the task i is done in a similar fashion, we start including in the set DESCENDANTS(i) all the successors of the node i, after it is done we include in the set all descendants of all the successors of the task i.

b. Implicit Enumeration

This approach requires the following steps:

1. Obtain the predecessors of all the tasks in the graph of constraints.

2. Obtain a legal sequence for the graph of constraints.

3. Evaluate the legal sequence obtained.

Figure 13 [Ref. 8] on page 53 illustrates the first level DFD for implicit enumeration.

B. GENERAL CONCEPT OF THE BRANCH AND BOUND STATIC SCHEDULING ALGORITHM

This algorithm also includes two steps to get the goal which is feasible or optimal for the static scheduler. The first step is generating the graph of constraints, the same as for the exhaustive enumeration alogrithm. The second step is getting the feasible or optimal scheduling by branch and bound techniques.

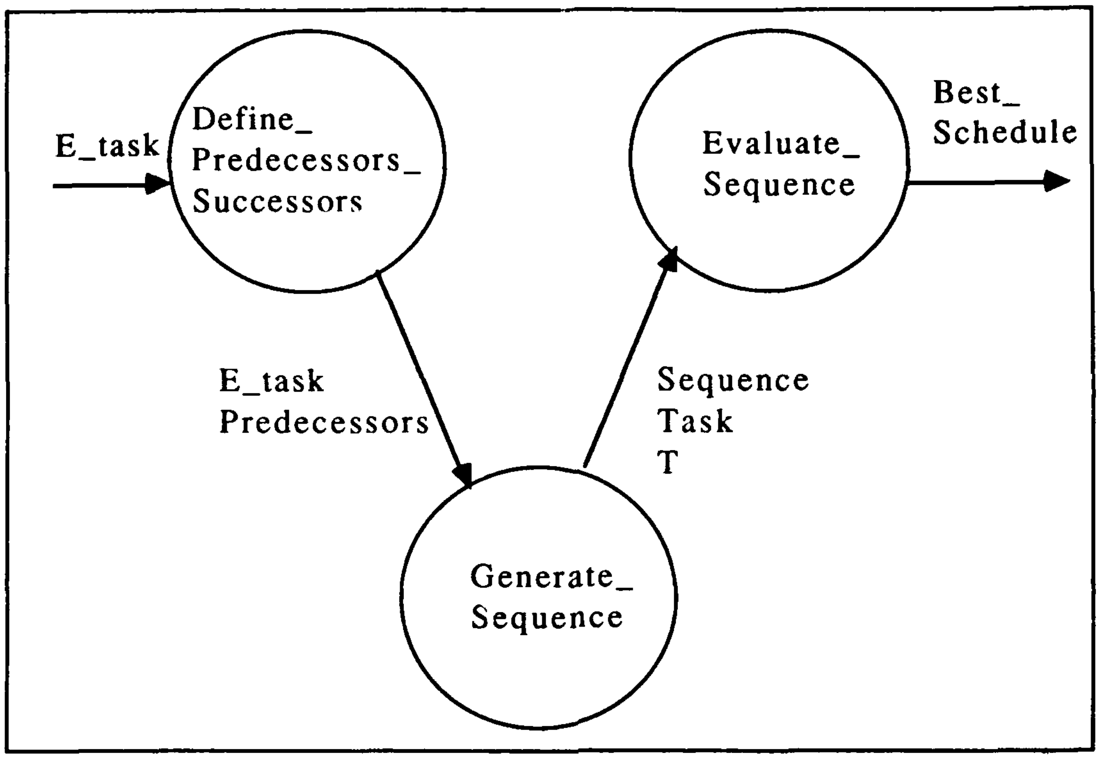


Figure 13 The First Level DFD for Implicit Enumeration

C. GETTING THE FEASIBLE OR OPTIMAL SCHEDULING BY BRANCH AND BOUND TECHNIQUES

The steps of the branch and bound static scheduling algorithm are desçribed as follows:

1. Scan the graph of constraints and get a node without any incoming edges, remove the first one and include it in the list.

2. Get the maximum cost of the scheduled tasks, assign to list_cost.

3. Get the maximum cost of the unscheduled tasks, assign to estimate_cost.

4. Get the maximum value of the list_cost and estimate_cost assign to maximum_cost.

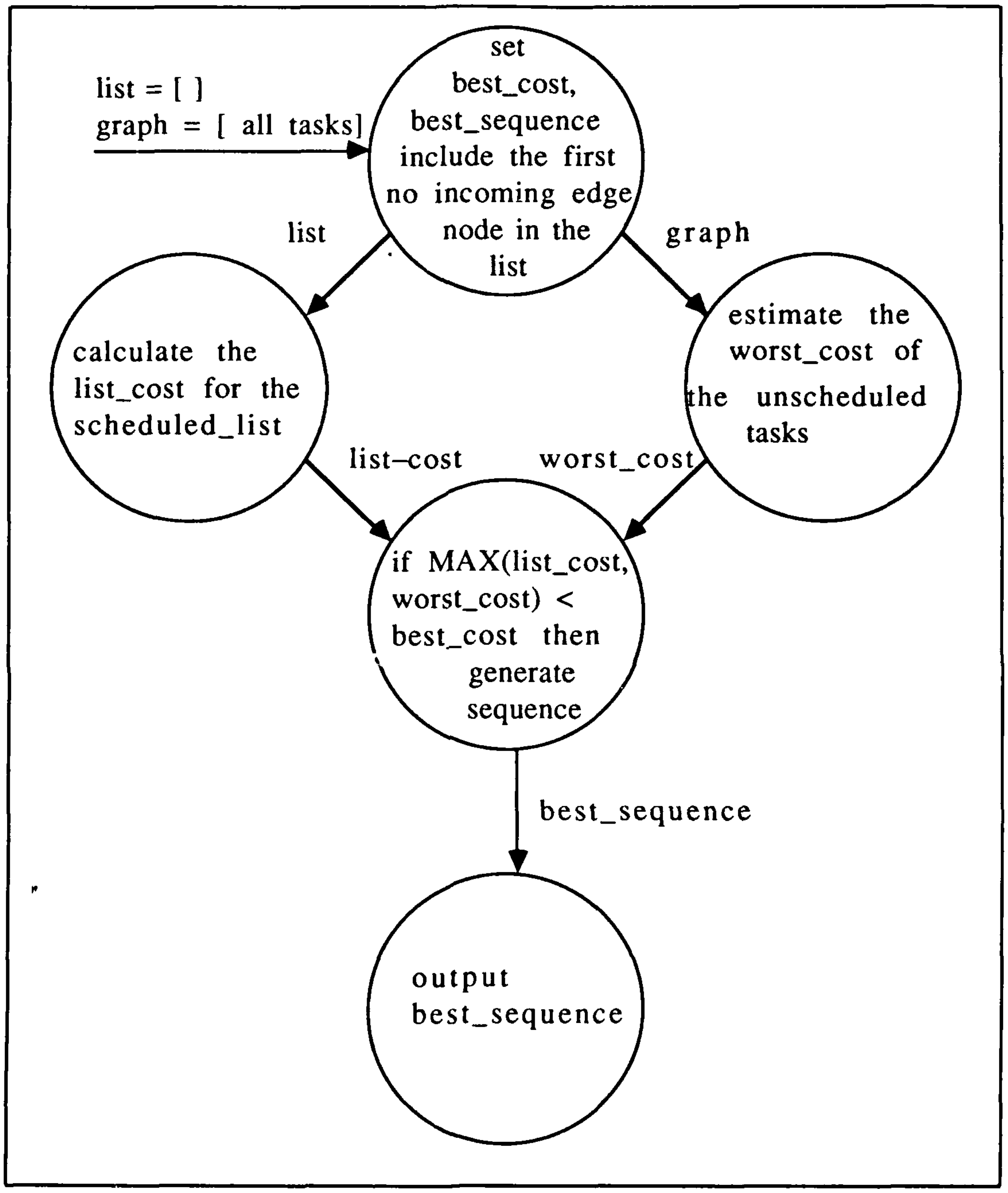

list = [ ]
graph = [all tasks]
set best_cost, best_sequence include the first no incoming edge node in the list
list
graph
calculate the list_cost for the scheduled_list
estimate the worst_cost of the unscheduled tasks
list–cost
worst_cost
if MAX(list_cost, worst_cost) < best_cost then generate sequence
best_sequence
output best_sequence

Figure 14 The first level DFD for branch and bound static scheduling

5. If the maximum_cost < best_cost then continue the search else try another branch.

6. If the search reaches a leaf node, then compare the list_cost against the best_cost. If the list_cost is less than the best_cost, then best_cost = list_cost and the best_sequence = list. If the best_cost <= 0 then the best_list is a feasible sequence and we can stop, else continue the search. At the end of the search if the best_cost > 0, then there is no feasible schedule. The best_sequence is the optimal sequence, and the cost is the best_cost.

Figure 14 on page 54 illustrates the first level DFD for branch and bound.

D. DATA STRUCTURES

There are 7 data type abstractions used in the current implementations. They are as follows:

Abstract data type	data structure
operators	array
list	array
status	array
queue	link_list
stack	link_list
op_precedence	matrix
task_precedence	matrix

Operators is a global data type that associates each operator with an id number. A list contains the scheduled tasks, while status records the status of each operator ('False' means it is not scheduled, 'true' means it is in the scheduled task list). Queue contains the nodes with no incoming edges in the graph and stack contains the last task's completion time for the list. Op_precedence is a two dimensional array, in which the length of both the columns and the rows are equal to the operator size. If op_precedence[i,j] = '1' then the i operator is the predecessor of the j operator, otherwise op_precedence[i,j] = '0'. Task_precedence[i,j] = '1' means that the task i is the predecessor of task j, otherwise task_precedence[i,j] = '0'.

E. "PUBLIC" PACKAGE

This package is share by the optimal static scheduling algorithm and the branch and bound static scheduling algorithm. It includes the following functions and procedures:

1. evaluate_lcm: calculates the least common multiple of all operators' periods.

    input: operators; operators' periods;

    output: the lcm of operators' periods (length of harmonic block)..

2. evaluate_gcd: calculate the greatest common divisor of two integer.

    input: two operators' periods;

    output: the gcd of this two operators' periods.

3. initialize_matrix: initialize the matrix of the precedence constraints of tasks.

    input: number of tasks; PSDL implementation graph;
    operator's periods

output: a matrix of precedence constraints.

4. define_task: generate a record which include the operator's id and instance.

input: task, operator, instance;

output: a record which include the operator's id and instance.

5. task_operator: get the task's operator's id.

input: record of operator's id and instance;

output: operator's id of a task.

6. task_instance: get the task's instance.

input: record of operator's id and instance;

output: task's instance.

7. precedence: decide if there is a precedence relationship between two operators or not.

input: two operators, the matrix of precedence constraints of operators.

output: a boolean value.

Details of the code are given in Appendix A.

F. DATA HANDLING ROUTINES

1. Exhaustive Enumeration

1). get_data: this procedure is used to read the data from the input file.

input: input file, APPENDIX A illustrates its format;

output: the array of operators, operators' periods, operators' maximum excute time(met), the precedence of operators.

2). get_total_task_number: this procedure calculates the total number of tasks.

input: operators , precedence of operators,
operators'periods;

output : total_number_of_tasks.

3). generate_precedence_graph: this procedure generates the precedence of all tasks.

input: total_number_of_tasks, operators, operators'
periods;

output: task_precedence which is a matrix '1'
represents that there exists a precedence
relationship, '0' represents otherwise.

4). include_task: this procedure is used to put a task at the end of a list.

input: a task, a link_list;

output: a link_list which include the task at the end.

5). add_task: this procedure is used to put a task in an array.

input: a task, an array;

output: an array which include the task.

6). remove_task: remove a task from an array.

input: a task, an array;

output: an array which deleted the task.

7). pop_stack: get the last task from a link_list.

input: a link_list;

output: a task that is removed from the end of the link_list.

8). get_queue: get the node(task), with no incoming edges from the precedence graph.

input: task_precedence;

output: an array which include all the no income node from the precedence constraints graph.

9). get_task: get the first task from a link_list.

input: link_list;

output: a task that is removed from the first of a link_list.

10). get_tardiness: get the tardiness from a specific task.

input: task, operators'periods, task_vec, operators' MET, task_precedence;

output: tardiness of the input task.

11). get_feasible_sequence: get the feasible sequence from a constraints graph of tasks.

input: task_precedence, task_vec, scheduled_list;

output: a scheduled_list.

2. Branch and Bound

1). get_ancestor: get ancestors which are not scheduled of a task.

input: parent_mtx, status;

output: a link_list which includes the ancestors which are not scheduled.

2). get_cost: get a list cost (the worst tardiness of a task of a list).

input: link_list; status;

output: list_cost which is the worst tardiness of a task of
the list.

3). get_least_cost: get the maximum estimate cost of unscheduled tasks.

input: status, task_size, completion time of list,
ancestors of tasks;

output: least_cost which is the maximum estimate cost of
all unscheduled tasks.

4). branch_bound: using branch and bound method get the optimal or feasible scheduled sequence of tasks.

input: list(scheduled tasks), status, task_size;

output: an array which includes the scheduled tasks,
sequence_cost which is the cost of the scheduled
tasks.

5). print_result: print the optimal sequence or feasible sequence, the sequence cost and the schedule of every task in the sequence.

input: an array, sequence_cost;

output: print out the scheduled tasks, and the cost of it;

The code for these routines is given in Appendix B.

G. THE COMPARISON OF BRANCH AND BOUND STATIC SCHEDULING AND THE EXHAUSTIVE ENUMERATION STATIC SCHEDULING

The explicit enumeration method first finds out all legal sequences which meet the precedence constraints, then tests if any of these legal sequences meets the timing constraints. If it does, then it is feasible sequence. The implicit enumeration method finds one legal sequence at a time and tests if it meets the timing constraints.

The explicit enumeration algorithm uses a lot of memory to store all the legal sequences. One of the advantages of the implicit enumeration algorithm is the saving of storage space because it stores just the best sequence. The advantage of the branch and bound algorithm is the saving of both storage and execution time. Since there is no need to expand the nodes which have worse estimate values then the best sequence (current one), it can avoid spending time to explore useless nodes.

Execution time and execution space are key factors in real-time scheduling. Branch and bound techniques can help us reach these goals. In addition, both explicit and implicit methods find out the feasible sequence; however, they cannot find out optimal sequence when there does not exist feasible sequence. The branch and bound algorithm solves this problem.

Table 1 illustrates the comparison of execution time of the exhaustive enumeration algorithm and the branch and bound algorithm.

The input and output for the example is given in Appendix C.

Example ID	Number of tasks	Execution time of exhaustive	Excution time of branch_bound
E 1	1 2	84.3 sec	38.0 sec
E 2	1 4	13848.4 sec	546.4 sec
E 3	1 4	6048.1 sec	488.8 sec

Table 1 The comparison of execution time of Branch and bound and exhaustive enumeration static scheduling

V. CONCLUSIONS AND SUGGESTIONS FOR FUTURE WORK

A. SUMMARY

This thesis has provided an introduction to two software engineering methodologies, the traditional life cycle and rapid prototyping. The rapid prototyping methodology is more efficient and less costly than the traditional software methodology. Real-time applications are important to computers and require the timing behavior of the system. We introduce the components of the Computer Aided Prototyping System (CAPS). One of the most critical of the components is the Static Scheduler.

We survey the previous research on static scheduling algorithms. We first introduce the general description of CAPS. The Prototyping System Description Language (PSDL) is designed for clarifying the requirements of complex real-time systems. We introduce six algorithms for static scheduling with different requirements.

In Chapter 3 we design a static scheduling algorithm using the branch and bound method. We first give the preliminary definition of some terms. The most critical part in this algorithm is how to find the estimate value of the unscheduled tasks. We give a heuristic method to estimate such cost. Following this new algorithm, we also give an example to illustrate it.

During the implementation, several different data structures have been tried to find the final result.

B. SUGGESTIONS FOR FURTHER RESEARCH

Our suggestions for future work in the area of scheduling algorithms are the following:

Refine the data structures used in the Static Scheduler for more efficient execution.

Refine the branch and bound technique by using a heuristic method to obtain a better initial value for the best_cost, and using the maximum tardiness in the preemptive schedule to get a better lower bound estimate.

Start a theoretical analysis to extend the CAPS system for the case of multiprocessors.

C. CONCLUSIONS

The goals of this thesis are to introduce the branch and bound static scheduling algorithm and implement the optimal static scheduling algorithm and the branch and bound algorithm. The main contribution of this work are the concepts of using branch and bound technique to find a feasible or optimal static scheduling for hard real-time systems, which saves time and storage space.

APPENDIX A
"PUBLIC" PACKAGE

```
with TEXT_IO;
use  TEXT_IO;

package PUBLIC is
  type VECTOR       is array(POSITIVE range <>) of INTEGER;

  type TASK_RECORD is record
    OPERATOR_ID : INTEGER;
    INSTANCE    : INTEGER;
  end record;

  type TASKS_VECTOR is array(POSITIVE range <>) of TASK_RECORD;

  type ID_NUMBER    is ('1', '0');

  type MATRIX       is array(POSITIVE range <>, POSITIVE range <>) of
   ID_NUMBER;

  type INT_MATRIX   is array(POSITIVE range <>, POSITIVE range <>) of
   INTEGER;

  -- caculate the least common multiple of all the operators that
  -- belongs to the set in analysis
  function EVALUATE_LCM(OPERATORS, OPERATORS_P : VECTOR) return INTEGER;

  -- caculate the greatest common divisor of the two periods
  function EVALUATE_GCD(X, Y : INTEGER) return INTEGER;

  -- caculate the number of tasks of an operator
  function OPERATOR_TASKS(OP           : INTEGER;
                          SIZE         : INTEGER;
                          OP_P         : INTEGER;
                          BLOCK_LENGTH : INTEGER) return INTEGER;

  -- initialize the matrix of the precedence constraints of tasks
  procedure INITIALIZE_MATRIX(TASK_PRECEDENCE : out MATRIX;
                              NUMBER_OF_TASKS : in INTEGER);

  -- define taks' operator and instance
  procedure DEFINE_TASK(TASKS       : in INTEGER;
                        OP          : in INTEGER;
                        INSTANCE    : in INTEGER;
                        TASK_VECTOR : in out TASKS_VECTOR);

  -- set '1' into the precedence constraints matrix if there are
  -- precedence relation occurs
  procedure SET_PRECEDENCE(TASK1, TASK2    : in POSITIVE;
                           TASK_PRECEDENCE : in out MATRIX);
  -- get the operator which the task belong it
  function TASK_OPERATOR(TASKS       : INTEGER;
                         TASK_VECTOR : TASKS_VECTOR) return INTEGER;

  -- get the instance which the task belong it
  function TASK_INSTANCE(TASKS       : INTEGER;
                         TASK_VECTOR : TASKS_VECTOR) return INTEGER;

  -- check the two operators have the precedence relation or not
  -- if yes return true else return false
  function PRECEDENCE(OP1, OP2             : INTEGER;
                      OPERATORS_PRECEDENCE : MATRIX) return BOOLEAN;
end PUBLIC;

package body PUBLIC is

  function EVALUATE_GCD(X, Y : INTEGER) return INTEGER is
    GCD : INTEGER;

  begin
    if Y > X then
      GCD := EVALUATE_GCD(Y, X);
    elsif (X mod Y) = 0 then
      GCD := Y;
    else
      GCD := EVALUATE_GCD(Y, X mod Y);
    end if;
    return GCD;
  end EVALUATE_GCD;


  function EVALUATE_LCM(OPERATORS, OPERATORS_P : VECTOR)
                       return INTEGER is
    LCM, P, N : INTEGER;

  begin
    LCM := 1;
    for I in 1 .. OPERATORS'LENGTH loop
      P := OPERATORS_P(I);
      LCM := (LCM * P) / EVALUATE_GCD(LCM, P);
    end loop;
    return LCM;
  end EVALUATE_LCM;


  function OPERATOR_TASKS(OP           : INTEGER;
                          SIZE         : INTEGER;
                          OP_P         : INTEGER;
                          BLOCK_LENGTH : INTEGER) return INTEGER is
    NUMBER_OF_TASKS : INTEGER;

  begin
    NUMBER_OF_TASKS := BLOCK_LENGTH / OP_P;
    return NUMBER_OF_TASKS;
```

```
end OPERATOR_TASKS;


procedure INITIALIZE_MATRIX(TASK_PRECEDENCE : out MATRIX;
                            NUMBER_OF_TASKS : in INTEGER) is

begin
  for I in 1 .. NUMBER_OF_TASKS loop
    for J in 1 .. NUMBER_OF_TASKS loop
      TASK_PRECEDENCE(I, J) := '0';
    end loop;
  end loop;
end INITIALIZE_MATRIX;


procedure DEFINE_TASK(TASKS       : in INTEGER;
                      OP          : in INTEGER;
                      INSTANCE    : in INTEGER;
                      TASK_VECTOR : in out TASKS_VECTOR) is

begin
  TASK_VECTOR(TASKS).OPERATOR_ID := OP;
  TASK_VECTOR(TASKS).INSTANCE := INSTANCE;
end DEFINE_TASK;
procedure SET_PRECEDENCE(TASK1, TASK2     : in POSITIVE;
                         TASK_PRECEDENCE : in out MATRIX) is

begin
  TASK_PRECEDENCE(TASK1, TASK2) := '1';
end SET_PRECEDENCE;


function TASK_OPERATOR(TASKS       : INTEGER;
                       TASK_VECTOR : TASKS_VECTOR) return INTEGER is
  OP : INTEGER;

begin
  OP := TASK_VECTOR(TASKS).OPERATOR_ID;
  return OP;
end TASK_OPERATOR;


function TASK_INSTANCE(TASKS       : INTEGER;
                       TASK_VECTOR : TASKS_VECTOR) return INTEGER is
  INSTANCE : INTEGER;

begin
  INSTANCE := TASK_VECTOR(TASKS).INSTANCE;
  return INSTANCE;
end TASK_INSTANCE;


function PRECEDENCE(OP1, OP2              : INTEGER;
                    OPERATORS_PRECEDENCE : MATRIX) return BOOLEAN is
    LINK : BOOLEAN := FALSE;

  begin
    if OPERATORS_PRECEDENCE(OP1, OP2) = '1' then
      LINK := TRUE;
    end if;
    return LINK;
  end PRECEDENCE;

end PUBLIC;
```

APPENDIX B
APPLICATION PROGRAMS

EXHAUSTIVE ENUMERATION

```
with TEXT_IO, PUBLIC, UNCHECKED_DEALLOCATION;

use  TEXT_IO, PUBLIC;

procedure FIND_SCHEDULE is

  package INT_IO is new INTEGER_IO(INTEGER);
  use  INT_IO;

  INF : FILE_TYPE;    -- input file
  NUMBER_OF_OPERATOR : INTEGER;


  procedure GET_SCHEDULE(NUMBER_OF_OP : in INTEGER) is

    package ID_IO is new ENUMERATION_IO(ID_NUMBER);
    use  ID_IO;

    -- operators is the set of the all operators
    -- operators_p is the period of a task
    -- op_met is the maximum excution time of a operator
    -- op_finish_within is the finish within time of a operator
    OPERATORS, OPERATORS_P, OP_MET, OP_FINISH_WITHIN :
     VECTOR(1 .. NUMBER_OF_OP);

    -- the matrix of number of tasks of every operator
    NUMBER_OF_TASKS : VECTOR(1 .. NUMBER_OF_OP);

    -- the total number of tasks of all operators
    TOTAL_NUMBER_OF_TASKS : INTEGER := 0;

    -- harmonic block length
    H_B_L : INTEGER;

    -- the matrix of the precedence constraints of the operator
    OP_PRECEDENCE : MATRIX(1 .. NUMBER_OF_OP, 1 .. NUMBER_OF_OP);
    -- read input file and get given data

    procedure GET_DATA is

      OPERATOR_PRECEDENCE : INT_MATRIX(1 .. NUMBER_OF_OP,
       1 .. NUMBER_OF_OP);
      -- m, n index
      M                    : INTEGER := 0;
      N                    : INTEGER;

    begin
  INITIALIZE_MATRIX(OP_PRECEDENCE, NUMBER_OF_OP);
  SKIP_LINE(INF);
  SKIP_LINE(INF);
  while not END_OF_FILE(INF) loop
    M := M + 1;
    -- get operator from input file
    GET(INF, OPERATORS(M));

    -- get the period of the operator which got from the input file
    GET(INF, OPERATORS_P(M));

    -- get the MET of the operator which got from the input file
    GET(INF, OP_MET(M));

    -- get the finish whithin time of the operator
    -- which got from the input file
    GET(INF, OP_FINISH_WITHIN(M));
    N := 0;
    while not END_OF_LINE(INF) loop
      N := N + 1;

      -- get the precedence constraints from the input file
      GET(INF, OPERATOR_PRECEDENCE(M, N));

      -- operator_precedence(m,n) /= 0 represents this is not
      -- the first (dummy) operator
      if OPERATOR_PRECEDENCE(M, N) /= 0 then
        SET_PRECEDENCE(OPERATOR_PRECEDENCE(M, N), OPERATORS(M),
         OP_PRECEDENCE);
      end if;
    end loop;
  end loop;
end GET_DATA;


-- caculate the number of tasks of every operator and
-- get the total number of tasks

procedure GET_TOTAL_TASK_NUMBER is

begin
  for I in 1 .. OPERATORS'LENGTH loop
    NUMBER_OF_TASKS(I) := OPERATOR_TASKS(OPERATORS(I), OPERATORS'LENGTH,
     OPERATORS_P(I), H_B_L);
    TOTAL_NUMBER_OF_TASKS := TOTAL_NUMBER_OF_TASKS +
     NUMBER_OF_TASKS(I);
  end loop;
end GET_TOTAL_TASK_NUMBER;
-- find the feasible or optimal sequence

procedure GET_TASK_SEQUENCE(TOTAL_NUMBER_OF_TASKS : in INTEGER) is

  type STATUS_VEC is array(INTEGER range <>) of BOOLEAN;
```

```
-- this matrix include operator and instance of every task
TASK_VECTOR      : TASKS_VECTOR(1 .. TOTAL_NUMBER_OF_TASKS);

-- matrix of the precedence constraints of the operator
-- the matrix of the precedence constraints of the operator
TASK_PRECEDENCE : MATRIX(1 .. TOTAL_NUMBER_OF_TASKS,
 1 .. TOTAL_NUMBER_OF_TASKS);

-- generate the graph of the precedence of tasks
-- task_precedence(i, j) = 0 means task i is the precedence of
-- task j

procedure GENERATE_PRECEDENCE_G is

-- the period of an operator
  P1, P2               : INTEGER;
  OP ID, OP_ID2        : INTEGER;
  INSTANCE, INSTANCE2 : INTEGER;

  -- define the operator and the instance of a task

  procedure DEFINE_TASK_OP_INS is

    T    : INTEGER := 1;     -- t represents task
    LAST : INTEGER;

  begin
                                 -- i represents the operator
    for I in 1 .. OPERATORS'LENGTH loop
      LAST := NUMBER_OF_TASKS(I) - 1;

      -- j represents the instance
      for J in 0 .. LAST loop
        DEFINE_TASK(T, I, J, TASK_VECTOR);
        T := T + 1;
      end loop;
    end loop;
  end DEFINE_TASK_OP_INS;
  -- generate the chains of the precedence of task
  -- if task i is the parent of task j then put
  -- task_precedence(i,j) = '1'

  procedure GENERATE_PRECEDENCE is

    T    : INTEGER := 1;     -- t represents task

    -- the tatal number of task of any operator minus one
    -- because the instance is begin from zero
    LAST : INTEGER;

  begin
    INITIALIZE_MATRIX(TASK_PRECEDENCE, TOTAL_NUMBER_OF_TASKS);
    -- i represents the operator
    for I in 1 .. OPERATORS'LENGTH loop
```

```
      LAST := NUMBER_OF_TASKS(I) - 1;

      -- j represents the instance
      for J in 0 .. LAST loop
        if J /= 0 then
          SET_PRECEDENCE(T - 1, T, TASK_PRECEDENCE);
        end if;
        T := T + 1;
      end loop;
    end loop;
  end GENERATE PRECEDENCE;


  -- the body of the procedure generate_precedence_g

begin
  DEFINE_TASK_OP_INS;
  GENERATE_PRECEDENCE;
  for I in 1 .. TOTAL_NUMBER_OF_TASKS loop
    OP_ID := TASK_OPERATOR(I, TASK_VECTOR);
    INSTANCE := TASK_INSTANCE(I, TASK_VECTOR);
    P1 := OPERATORS_P(OP_ID);
    for J in 1 .. TOTAL_NUMBER_OF_TASKS loop
      OP_ID2 := TASK_OPERATOR(J, TASK_VECTOR);
      INSTANCE2 := TASK_INSTANCE(J, TASK_VECTOR);
      P2 := OPERATORS_P(OP_ID2);
      if OP_ID /= OP_ID2 then
        if P1 * INSTANCE = P2 * INSTANCE2 and PRECEDENCE(OP_ID, OP_ID2,
         OP_PRECEDENCE) then
          SET_PRECEDENCE(I, J, TASK_PRECEDENCE);
        end if;
      end if;
    end loop;
  end loop;
end GENERATE_PRECEDENCE_G;
-- use to get the scheduled sequence for the real_time systems

procedure GET_SEQUENCE(TASK_PRECEDENCE : in MATRIX;
                       TASK_SIZE       : in INTEGER) is

  type LINK_RECORD;
  type TASK_LINK   is access LINK_RECORD;
  type LINK_RECORD is record
    NEXT : TASK_LINK;
    P    : INTEGER;
  end record;
  type LINK_MATRIX is array(INTEGER range <>) of TASK_LINK;

  -- the check of the tasks status
  STATUS          : STATUS_VEC(1 .. TASK_SIZE)       :=
   (others => FALSE);
  LIST            : VECTOR(1 .. TASK_SIZE)           :=
   (others => 0);
  LIST_STATUS     : STATUS_VEC(1 .. TASK_SIZE)       :=
 (others => FALSE);
LAST_COMPLETION : INTEGER                           := 0;
PHASE           : VECTOR(1 .. OPERATORS'LENGTH);
-- check the task's operator has been scheduled already
PHASE_DONE      : STATUS_VEC(1 .. OPERATORS'LENGTH) :=
 (others => FALSE);
-- the cost of sequence
SEQ_COST        : INTEGER;
-- the sequence which has the best cost
BEST_SEQUENCE   : VECTOR(1 .. TASK_SIZE);
START_TIME      : VECTOR(1 .. TASK_SIZE);
END_TIME        : VECTOR(1 .. TASK_SIZE);
TARDINESS       : INTEGER                           :=
 INTEGER'LAST;

-- use to initialize the link list

procedure INITIALIZE_POINTER(POINTERS : in out LINK_MATRIX;
                             SIZE     : in INTEGER) is

begin
  for I in 1 .. SIZE loop
    POINTERS(I) := null;
  end loop;
end INITIALIZE_POINTER;


-- put a task into a link list

procedure INCLUDE_TASK(TASKS : in INTEGER;
                       LIST  : in out TASK_LINK) is

begin
  if LIST = null then
    LIST := new LINK_RECORD'(null, TASKS);
  else
    INCLUDE_TASK(TASKS, LIST.NEXT);
  end if;
end INCLUDE_TASK;
-- write a vector

procedure WRITE_VECTOR(SIZE     : in INTEGER;
                       DATA_VEC : in VECTOR) is

begin
  for I in 1 .. SIZE loop
    PUT(DATA_VEC(I), WIDTH => 4);
    if I mod 20 = 0 then
      NEW_LINE;
    end if;
  end loop;
end WRITE_VECTOR;
```

```
-- get the number income edge of every node's (task)
-- of the graph of precedence constraints then put in a queue

function GET_QUEUE(NUMBER   : INTEGER;
                   STATUS_V : STATUS_VEC) return TASK_LINK is

  IN_DEGREE  : INTEGER;
  QUEUE_LIST : TASK_LINK := null;

  -- get the task's income edge number

  function GET_INDEG(I              : INTEGER;
                     STATUS_VECTOR : STATUS_VEC)
                      return INTEGER is

    DEGREE : INTEGER := 0;

  begin
    for J in 1 .. TASK_SIZE loop
      if not STATUS_VECTOR(J) and TASK_PRECEDENCE(J, I) =
       '1' then
        DEGREE := DEGREE + 1;
      end if;
    end loop;
    return DEGREE;
  end GET_INDEG;

  -- the body of procedure get_queue

begin
  for I in 1 .. TASK_SIZE loop
    IN_DEGREE := GET_INDEG(I, STATUS_V);
    if IN_DEGREE = 0 and STATUS_V(I) = FALSE then
      INCLUDE_TASK(I, QUEUE_LIST);
  • end if;
  end loop;
  return QUEUE_LIST;
end GET_QUEUE;
-- get first task from a link list

procedure GET_TASK(T_LIST : in out TASK_LINK;
                   TASKS  : out INTEGER) is

  TEMP : TASK_LINK := null;
  procedure FREE is new UNCHECKED_DEALLOCATION(LINK_RECORD,
                                               TASK_LINK);
begin
  TASKS := T_LIST.P;
  TEMP := T_LIST;
  T_LIST := T_LIST.NEXT;
  FREE(TEMP);
end GET_TASK;
```

```
-- get the phase of a task

function GET_S_PHASE(OP,
                     LAST_COMPLETION      : INTEGER) return INTEGER is

  PHASE_VALUE : INTEGER;

begin
  PHASE_VALUE := LAST_COMPLETION;
  PHASE_DONE(OP) := TRUE;
  return PHASE_VALUE;
end GET_S_PHASE;


-- get tardiness of the task

procedure GET_TARDINESS(ELEMENT          : in INTEGER;
                        TARDINESS        : out INTEGER;
                        LAST_COMPLETION : in out INTEGER) is

  COMPLETION : INTEGER;
  DEADLINE   : INTEGER;

  -- get the earliest start time of the task from a sequence

  function GET_EARLIEST_START(ELEMENT, OP,
                              LAST_COMPLETION              : INTEGER)
                              return INTEGER is

    EARLIEST_START_VALUE, PHASE_VALUE : INTEGER := 0;
    INSTANCE                          : INTEGER;

  begin
    INSTANCE := TASK_INSTANCE(ELEMENT, TASK_VECTOR);
    if not PHASE_DONE(OP) then
      PHASE(OP) := GET_S_PHASE(OP, LAST_COMPLETION);
    end if;
    EARLIEST_START_VALUE := INSTANCE * OPERATORS_P(OP) +
     PHASE(OP);
    return EARLIEST_START_VALUE;
  end GET_EARLIEST_START;
  -- get the deadline time of a task from a sequence

  function GET_DEADLINE(ELEMENT, LAST_COMPLETION : INTEGER)
                        return INTEGER is

    DEADLINE_VALUE, TEMP, EARLIEST_START_VALUE : INTEGER;

    OP                                          : INTEGER;

  begin
    OP := TASK_OPERATOR(ELEMENT, TASK_VECTOR);
    EARLIEST_START_VALUE := GET_EARLIEST_START(ELEMENT, OP,
     LAST_COMPLETION);
    TEMP := EARLIEST_START_VALUE + OP_FINISH_WITHIN(OP);
    if TEMP > H_B_L then
      DEADLINE_VALUE := H_B_L;
    else
      DEADLINE_VALUE := TEMP;
    end if;
    return DEADLINE_VALUE;
  end GET_DEADLINE;


  -- get the start time of a task from a sequence

  function GET_START(LAST_COMPLETION, ELEMENT, OP : INTEGER)
                     return INTEGER is

    START_VALUE, EARLIEST_START_VALUE : INTEGER;

  begin
    EARLIEST_START_VALUE := GET_EARLIEST_START(ELEMENT, OP,
     LAST_COMPLETION);
    if LAST_COMPLETION > EARLIEST_START_VALUE then
      START_VALUE := LAST_COMPLETION;
    else
      START_VALUE := EARLIEST_START_VALUE;
    end if;
    START_TIME(ELEMENT) := START_VALUE;
    return START_VALUE;
  end GET_START;


  -- get the completion time of a task from a sequence

  function GET_COMPLETION(LAST_COMPLETION, ELEMENT : INTEGER)
                          return INTEGER is

    COMPLETION_VALUE, START_VALUE, OPERATOR : INTECER;

  begin
    OPERATOR := TASK_OPERATOR(ELEMENT, TASK_VECTOR);
    START_VALUE := GET_START(LAST_COMPLETION, ELEMENT,
     OPERATOR);
    COMPLETION_VALUE := START_VALUE + OP_MET(OPERATOR);
    END_TIME(ELEMENT) := COMPLETION_VALUE;
    return COMPLETION_VALUE;
  end GET_COMPLETION;
  -- the body of the procedure get_tardiness

begin
  COMPLETION := GET_COMPLETION(LAST_COMPLETION, ELEMENT);
  DEADLINE := GET_DEADLINE(ELEMENT, LAST_COMPLETION);
  TARDINESS := COMPLETION - DEADLINE;
  LAST_COMPLETION := COMPLETION;
end GET_TARDINESS;
```

```
-- get a task from an array

procedure GET_LIST_TASK(LIST        : in VECTOR;
                        LIST_STATUS : in out STATUS_VEC;
                        TASK_VALUE  : out INTEGER) is

begin
  for I in 1 .. TASK_SIZE loop
    if LIST_STATUS(I) then
      TASK_VALUE := LIST(I);
      LIST_STATUS(I) := FALSE;
      exit;
    end if;
  end loop;
end GET_LIST_TASK;


-- remove a task from an array

procedure REMOVE_TASK(LIST_STATUS : in out STATUS_VEC) is

begin
  for I in reverse 1 .. TASK_SIZE loop
    if LIST_STATUS(I) then
      LIST_STATUS(I) := FALSE;
      exit;
    end if;
  end loop;
end REMOVE_TASK;

-- put a task into an array

procedure ADD_TASK(TASKS       : in INTEGER;
                   LIST        : in out VECTOR;
                   LIST_STATUS : in out STATUS_VEC) is

begin
  for I in 1 .. TASK_SIZE loop
    if not LIST_STATUS(I) then
      LIST(I) := TASKS;
      LIST_STATUS(I) := TRUE;
      exit;
    end if;
  end loop;
end ADD_TASK;
-- compare two value and return the maximum value

function MAX(LEFT : INTEGER; RIGHT : INTEGER) return INTEGER is

begin
  if LEFT > RIGHT then
    return LEFT;
  else
```

```
    return RIGHT;
  end if;
end MAX;

procedure GET_FEASIBLE_SEQUENCE(SEQ_LIST          : in VECTOR;
                                LIST_FLAG,
                                STATUS            : in STATUS_VEC;
                                NUMBER            : in INTEGER) is

  LIST         : VECTOR(1 .. TASK_SIZE)     := SEQ_LIST;
  LIST_STATUS  : STATUS_VEC(1 .. TASK_SIZE) := LIST_FLAG;
  STATUS_V     : STATUS_VEC(1 .. TASK_SIZE) := STATUS;
  TASKS        : INTEGER;
  TASK_VALUE   : INTEGER;
  QUEUE        : TASK_LINK                  := null;
  CHECK        : INTEGER                    := 0;
  IN_DEGREE    : INTEGER;

begin
  if TARDINESS <= 0 then
    return;
  end if;
  QUEUE := GET_QUEUE(NUMBER, STATUS_V);
  if QUEUE = null then
    for I in 1 .. TASK_SIZE loop
      GET_LIST_TASK(LIST, LIST_STATUS, TASK_VALUE);
      GET_TARDINESS(TASK_VALUE, TARDINESS, LAST_COMPLETION);
      exit when TARDINESS > 0;
    end loop;
    SEQ_COST := TARDINESS;
    if TARDINESS <= 0 then
      BEST_SEQUENCE := LIST;
      return;
    end if;

    PHASE_DONE := (others => FALSE);
    LAST_COMPLETION := 0;
    LIST_STATUS := (others => FALSE);
  else
    while QUEUE /= null loop
      GET_TASK(QUEUE, TASKS);
      ADD_TASK(TASKS, LIST, LIST_STATUS);
      STATUS_V(TASKS) := TRUE;
      GET_FEASIBLE_SEQUENCE(LIST, LIST_STATUS, STATUS_V,
       NUMBER);
      REMOVE_TASK(LIST_STATUS);
      STATUS_V(TASKS) := FALSE;
    end loop;
  end if;
end GET_FEASIBLE_SEQUENCE;
-- decide the sequence is feasible or optimal and print it out

procedure PRINT_RESULT(BEST_SEQ : in VECTOR) is
```

```
        begin
          if SEQ_COST <= 0 then
            PUT_LINE(" The feasible sequence is ");
            WRITE_VECTOR(TASK_SIZE, BEST_SEQUENCE);
            NEW_LINE(2);
            PUT_LINE("         TASK   START TIME  END TIME");
            PUT_LINE("        ------  ---------- ----------");
            for I in 1 .. TASK_SIZE loop
              PUT(BEST_SEQUENCE(I));
              PUT(START_TIME(BEST_SEQUENCE(I)));
              PUT(END_TIME(BEST_SEQUENCE(I)));
              NEW_LINE;
            end loop;

          else
            PUT_LINE(" There is no feasible sequence ");
            NEW_LINE(2);
          end if;
        end PRINT_RESULT;

        -- the body of procedure get_sequence

      begin
        GET_FEASIBLE_SEQUENCE(LIST, LIST_STATUS, STATUS, TASK_SIZE);
        PRINT_RESULT(BEST_SEQUENCE);
      end GET_SEQUENCE;

      -- the body of procedure get_task_sequence

    begin
      GENERATE_PRECEDENCE_G;
      GET_SEQUENCE(TASK_PRECEDENCE, TOTAL_NUMBER_OF_TASKS);
    end GET_TASK_SEQUENCE;

    -- the body of procedure get_schedule

  begin
    GET_DATA;
    H_B_L := EVALUATE_LCM(OPERATORS, OPERATORS_P);
    GET_TOTAL_TASK_NUMBER;
    GET_TASK_SEQUENCE(TOTAL_NUMBER_OF_TASKS);
  end GET_SCHEDULE;

  -- the body of main procedure

begin
  OPEN(INF, MODE => IN_FILE, NAME => "input_f");
  SKIP_LINE(INF);
  GET(INF, NUMBER_OF_OPERATOR);
  SKIP_LINE(INF);
  GET_SCHEDULE(NUMBER_OF_OPERATOR);
  CLOSE(INF);
end FIND_SCHEDULE;
```

BRANCH AND BOUND

```
with TEXT_IO, PUBLIC, UNCHECKED_DEALLOCATION;

use  TEXT_IO, PUBLIC;

procedure FIND_SCHEDULE is

  package INT_IO is new INTEGER_IO(INTEGER);
  use  INT_IO;

  INF : FILE_TYPE;     -- input file
  NUMBER_OF_OPERATOR : INTEGER;


  procedure GET_SCHEDULE(NUMBER_OF_OP : in INTEGER) is

    package ID_IO is new ENUMERATION_IO(ID_NUMBER);
    use  ID_IO;

    -- operators is the set of the all operators
    -- period is the period of a task
    -- op_met is the maximum excution time of a operator
    -- op_finish_within is the finish within time of a operator
    OPERATORS, PERIOD, OP_MET, OP_FINISH_WITHIN :
     VECTOR(1 .. NUMBER_OF_OP);

    -- the matrix of number of tasks of an operator
    NUMBER_OF_TASKS : VECTOR(1 .. NUMBER_OF_OP);

    -- the total number of tasks of all operators
    TOTAL_NUMBER_OF_TASKS : INTEGER := 0;

    -- harmonic block length
    H_B_L : INTEGER;

    -- the matrix of the precedence constraints of the operator
    OP_PRECEDENCE : MATRIX(1 .. NUMBER_OF_OP, 1 .. NUMBER_OF_OP);
    -- read input file and get given data

    procedure GET_DATA is

      OPERATOR_PRECEDENCE : INT_MATRIX(1 .. NUMBER_OF_OP,
       1 .. NUMBER_OF_OP);
      -- m, n index
      M                     : INTEGER := 0;
      N                     : INTEGER;
     begin
      INITIALIZE_MATRIX(OP_PRECEDENCE, NUMBER_OF_OP);
      SKIP_LINE(INF);
      SKIP_LINE(INF);
      while not END_OF_FILE(INF) loop
        M := M + 1;
        -- get operator from input file
    GET(INF, OPERATORS(M));

    -- get the period of the operator which got from the input file
    GET(INF, PERIOD(M));

    -- get the MET of the operator which got from the input file
    GET(INF, OP_MET(M));

    -- get the finish whithin time of the operator
    -- which got from the input file
    GET(INF, OP_FINISH_WITHIN(M));
    N := 0;
    while not END_OF_LINE(INF) loop
      N := N + 1;

      -- get the precedence constraints from the input file
      GET(INF, OPERATOR_PRECEDENCE(M, N));

      -- operator_precedence(m,n) /= 0 represents this is not
      -- the first (dummy) operator
      if OPERATOR_PRECEDENCE(M, N) /= 0 then
        SET_PRECEDENCE(OPERATOR_PRECEDENCE(M, N), OPERATORS(M),
         OP_PRECEDENCE);
      end if;
    end loop;
  end loop;
end GET_DATA;


-- caculate the number of tasks of every operator and
-- get the total number of tasks

procedure GET_TOTAL_TASK_NUMBER is

begin
  for I in 1 .. OPERATORS'LENGTH loop
    NUMBER_OF_TASKS(I) := OPERATOR_TASKS(OPERATORS(I), OPERATORS'LENGTH,
     PERIOD(I), H_B_L);
    TOTAL_NUMBER_OF_TASKS := TOTAL_NUMBER_OF_TASKS +
     NUMBER_OF_TASKS(I);
  end loop;
end GET_TOTAL_TASK_NUMBER;
-- find the feasible or optimal sequence

procedure GET_TASK_SEQUENCE(TOTAL_NUMBER_OF_TASKS : in INTEGER) is

  type STATUS_VEC is array(INTEGER range <>) of BOOLEAN;

  -- this matrix include operator ID and instance ID of every task
  TASK_VECTOR     : TASKS_VECTOR(1 .. TOTAL_NUMBER_OF_TASKS);

  -- matrix of the precedence constraints of the operator
  -- the matrix of the precedence constraints of the operator
  TASK_PRECEDENCE : MATRIX(1 .. TOTAL_NUMBER_OF_TASKS,
 1 .. TOTAL_NUMBER_OF_TASKS);

-- generate the graph of the precedence of tasks
-- task_precedence(i, j) = 0 means task i is the precedence of
-- task j

procedure GENERATE_PRECEDENCE_GRAPH is

-- the period of an operator
  P1, P2              : INTEGER;
  OP_ID, OP_ID2       : INTEGER;
  INSTANCE, INSTANCE2 : INTEGER;

  -- define the operator and the instance of a task

  procedure DEFINE_TASK_OP_INS is

    T    : INTEGER := 1;     -- t represents task
    LAST : INTEGER;

  begin
                                -- i represents the operator
    for I in 1 .. OPERATORS'LENGTH loop
      LAST := NUMBER_OF_TASKS(I) - 1;

      -- j represents the instance
      for J in 0 .. LAST loop
        DEFINE_TASK(T, I, J, TASK_VECTOR);
        T := T + 1;
      end loop;
    end loop;
  end DEFINE_TASK_OP_INS;
  -- generate the chains of the precedence of task
  -- if task i is the parent of task j then put
  --'task_precedence(i,j) = '1'

  procedure GENERATE_PRECEDENCE is

    T    : INTEGER := 1;     -- t represents task

    -- the tatal number of task of any operator minus one
    -- because the instance is begin from zero
    LAST : INTEGER;

  begin
    INITIALIZE_MATRIX(TASK_PRECEDENCE, TOTAL_NUMBER_OF_TASKS);
    -- i represents the operator
    for I in 1 .. OPERATORS'LENGTH loop
      LAST := NUMBER_OF_TASKS(I) - 1;

      -- j represents the instance
      for J in 0 .. LAST loop
        if J /= 0 then
          SET_PRECEDENCE(T - 1, T, TASK_PRECEDENCE);
        end if;
        T := T + 1;
      end loop;
    end loop;
  end GENERATE_PRECEDENCE;

  -- the body of the procedure generate_precedence_graph

begin
  DEFINE_TASK_OP_INS;
  GENERATE_PRECEDENCE;
  for I in 1 .. TOTAL_NUMBER_OF_TASKS loop
    OP_ID := TASK_OPERATOR(I, TASK_VECTOR);
    INSTANCE := TASK_INSTANCE(I, TASK_VECTOR);
    P1 := PERIOD(OP_ID);
    for J in 1 .. TOTAL_NUMBER_OF_TASKS loop
      OP_ID2 := TASK_OPERATOR(J, TASK_VECTOR);
      INSTANCE2 := TASK_INSTANCE(J, TASK_VECTOR);
      P2 := PERIOD(OP_ID2);
      if OP_ID /= OP_ID2 then
        if P1 * INSTANCE = P2 * INSTANCE2 and PRECEDENCE(OP_ID, OP_ID2,
         OP_PRECEDENCE) then
          SET_PRECEDENCE(I, J, TASK_PRECEDENCE);
        end if;
      end if;
    end loop;
  end loop;
end GENERATE_PRECEDENCE_GRAPH;
-- use to get the scheduled sequence for the real_time systems

procedure GET_SEQUENCE(TASK_PRECEDENCE : in MATRIX;
                       TASK_SIZE       : in INTEGER) is

  type LINK_RECORD;
  type TASK_LINK   is access LINK_RECORD;
  type LINK_RECORD is record
    NEXT : TASK_LINK;
    P    : INTEGER;
  end record;
  type LINK_MATRIX is array(INTEGER range <>) of TASK_LINK;

  -- the check of the tasks status
  STATUS         : STATUS_VEC(1 .. TASK_SIZE)  :=  (others => FALSE);
  LIST           : VECTOR(1 .. TASK_SIZE)      :=  (others => 0);
  LIST_STATUS    : STATUS_VEC(1 .. TASK_SIZE)  :=  (others => FALSE);

  -- the matrix which store the parent of the task
  PARENT_MTX     : LINK_MATRIX(1 .. TASK_SIZE);
  PARENT_NUMBER  : VECTOR(1 .. TASK_SIZE)      :=  (others => 0);
  -- the stack store the task's completion time
  -- the dummy task is given the value 0
  STACK          : TASK_LINK                   :=  null;
  -- the cost stack store the cost of all scheduled tasks
  COST_STACK     : TASK_LINK                   :=  null;
```

```
-- the last completion time
LAST_COMPLETION   : INTEGER        := 0;
PHASE        : VECTOR(1 .. OPERATORS'LENGTH);
-- check the task's operator has been scheduled already
PHASE_DONE   : STATUS_VEC(1 .. OPERATORS'LENGTH) :=(others => FALSE);
-- the cost of a sequence
SEQ_COST     : INTEGER              :=  INTEGER'FIRST;
-- the best(lower) cost of all found sequence
BEST_COST    : INTEGER              :=  INTEGER'LAST;
-- the sequence which has the best cost
BEST_SEQUENCE                : VECTOR(1 .. TASK_SIZE);
BEST_START_TIME, START_TIME : VECTOR(1 .. TASK_SIZE);
BEST_END_TIME, END_TIME     : VECTOR(1 .. TASK_SIZE);

procedure FREE is new UNCHECKED_DEALLOCATION(LINK_RECORD,
                                             TASK_LINK);

-- use to initialize the link list

procedure INITIALIZE_POINTER(POINTERS : in out LINK_MATRIX;
                             SIZE     : in INTEGER) is

begin
  for I in 1 .. SIZE loop
    POINTERS(I) := null;
  end loop;
end INITIALIZE_POINTER;
-- put a task into a link list

procedure INCLUDE_TASK(TASKS : in INTEGER;
                       LIST  : in out TASK_LINK) is

begin
  if LIST = null then
  , LIST := new LINK_RECORD'(null, TASKS);
  else
    INCLUDE_TASK(TASKS, LIST.NEXT);
  end if;
end INCLUDE_TASK;


procedure ADD_TASK(TASKS       : in INTEGER;
                   LIST        : in out VECTOR;
                   LIST_STATUS : in out STATUS_VEC) is

begin
  for I in 1 .. TASK_SIZE loop
    if not LIST_STATUS(I) then
      LIST(I) := TASKS;
      LIST_STATUS(I) := TRUE;
      exit;
    end if;
  end loop;
end ADD_TASK;
```

```
-- get the last task from a link list

function POP_STACK(LIST : TASK_LINK) return INTEGER is

  TEMP_VALUE : INTEGER;
begin
  if LIST.NEXT = null then
    TEMP_VALUE := LIST.P;
  else
    TEMP_VALUE := POP_STACK(LIST.NEXT);
  end if;
  return TEMP_VALUE;
end POP_STACK;


-- remove the last task from a link list

procedure REMOVE_TASK(LIST_STATUS : in out STATUS_VEC) is

begin
  for I in reverse 1 .. TASK_SIZE loop
    if LIST_STATUS(I) then
      LIST_STATUS(I) := FALSE;
      exit;
    end if;
  end loop;
end REMOVE_TASK;
procedure REMOVE_STACK(LIST : in out TASK_LINK) is

  TEMP : TASK_LINK := null;

begin
  if LIST /= null and LIST.NEXT /= null then
    REMOVE_STACK(LIST.NEXT);
  else
    TEMP := LIST;
    LIST := null;
    FREE(TEMP);
  end if;
end REMOVE_STACK;

-- write a vector out

procedure WRITE_VECTOR(SIZE     : in INTEGER;
                       DATA_VEC : in VECTOR) is

begin
  for I in 1 .. SIZE loop
    PUT(DATA_VEC(I), WIDTH => 4);
    if I mod 20 = 0 then
      NEW_LINE;
    end if;
  end loop;
end WRITE_VECTOR;


-- get the number no income edge of every node's (task)
-- of the graph of precedence constraints then put in a queue

function GET_QUEUE(NUMBER   : INTEGER;
                   STATUS_V : STATUS_VEC) return TASK_LINK is

  IN_DEGREE  : INTEGER;
  QUEUE_LIST : TASK_LINK := null;

  -- get the task's income edge number

  function GET_INDEG(I             : INTEGER;
                     STATUS_VECTOR : STATUS_VEC)
                      return INTEGER is

    DEGREE : INTEGER := 0;

  begin
    for J in 1 .. TASK_SIZE loop
      if not STATUS_VECTOR(J) and TASK_PRECEDENCE(J, I) =
       '1' then
        DEGREE := DEGREE + 1;
      end if;
    end loop;
    return DEGREE;
  end GET_INDEG;
  -- the body of procedure get_queue

begin
  for I in 1 .. TASK_SIZE loop
    IN_DEGREE := GET_INDEG(I, STATUS_V);
    if IN_DEGREE = 0 and STATUS_V(I) = FALSE then
      INCLUDE_TASK(I, QUEUE_LIST);
    end if;
  end loop;
  return QUEUE_LIST;
end GET_QUEUE;


-- get first task from a link list

procedure GET_TASK(T_LIST : in out TASK_LINK;
                   TASKS  : out INTEGER) is

  TEMP : TASK_LINK := null;
begin
  TASKS := T_LIST.P;
  TEMP := T_LIST;
  T_LIST := T_LIST.NEXT;
  FREE(TEMP);
end GET_TASK;
```

```
-- get the phase of the task which is included in the scheduled
-- list

function GET_SCHEDULED_PHASE(OP, LAST_COMPLETION : INTEGER)
                              return INTEGER is

  PHASE_VALUE : INTEGER;

begin
  PHASE_VALUE := LAST_COMPLETION;
  PHASE_DONE(OP) := TRUE;
  return PHASE_VALUE;
end GET_SCHEDULED_PHASE;


-- get the phase of the task which unscheduled

function GET_UNSCHEDULED_PHASE(ELEMENT, OP : in INTEGER)
                                return INTEGER is

  PHASE_VALUE : INTEGER;

begin
  PHASE_VALUE := PERIOD(OP);
  return PHASE_VALUE;
end GET_UNSCHEDULED_PHASE;

-- get tardiness of the task

procedure GET_TARDINESS(ELEMENT         : in INTEGER;
                        TARDINESS       : out INTEGER;
                        LAST_COMPLETION : in out INTEGER;
                        STACK           : in out TASK_LINK) is

  COMPLETION : INTEGER;
  DEADLINE   : INTEGER;

  -- get the earliest start time of the task from a sequence

  function GET_EARLIEST_START(ELEMENT, OP,
                              LAST_COMPLETION              : INTEGER)
                              return INTEGER is

    EARLIEST_START_VALUE, PHASE_VALUE : INTEGER := 0;
    INSTANCE                          : INTEGER;

  begin
    INSTANCE := TASK_INSTANCE(ELEMENT, TASK_VECTOR);
    if not PHASE_DONE(OP) then
      PHASE(OP) := GET_SCHEDULED_PHASE(OP, LAST_COMPLETION);
    end if;
    EARLIEST_START_VALUE := INSTANCE * PERIOD(OP) + PHASE(OP);
  return EARLIEST_START_VALUE;
end GET_EARLIEST_START;


-- get the deadline time of a task from a sequence

function GET_DEADLINE(ELEMENT, LAST_COMPLETION : INTEGER)
                      return INTEGER is

  DEADLINE_VALUE, TEMP, EARLIEST_START_VALUE : INTEGER;

  OP                                         : INTEGER;

begin
  OP := TASK_OPERATOR(ELEMENT, TASK_VECTOR);
  EARLIEST_START_VALUE := GET_EARLIEST_START(ELEMENT, OP,
   LAST_COMPLETION);
  TEMP := EARLIEST_START_VALUE + OP_FINISH_WITHIN(OP);
  if TEMP > H_B_L then
    DEADLINE_VALUE := H_B_L;
  else
    DEADLINE_VALUE := TEMP;
  end if;
  return DEADLINE_VALUE;
end GET_DEADLINE;


-- get the start time of a task from a sequence

function GET_START(LAST_COMPLETION, ELEMENT, OP : INTEGER)
                   return INTEGER is

  START_VALUE, EARLIEST_START_VALUE : INTEGER;
begin
  EARLIEST_START_VALUE := GET_EARLIEST_START(ELEMENT, OP,
   LAST_COMPLETION);
  if LAST_COMPLETION > EARLIEST_START_VALUE then
    START_VALUE := LAST_COMPLETION;
  else
    START_VALUE := EARLIEST_START_VALUE;
  end if;
  START_TIME(ELEMENT) := START_VALUE;
  return START_VALUE;
end GET_START;


-- get the completion time of a task from a sequence

function GET_COMPLETION(LAST_COMPLETION, ELEMENT : INTEGER)
                        return INTEGER is

  COMPLETION_VALUE, START_VALUE, OPERATOR : INTEGER;

begin
```

```
    OPERATOR := TASK_OPERATOR(ELEMENT, TASK_VECTOR);
    START_VALUE := GET_START(LAST_COMPLETION, ELEMENT,
     OPERATOR);
    COMPLETION_VALUE := START_VALUE + OP_MET(OPERATOR);
    END_TIME(ELEMENT) := COMPLETION_VALUE;
    return COMPLETION_VALUE;
  end GET_COMPLETION;

  -- the body of the procedure get_tardiness

begin
  COMPLETION := GET_COMPLETION(LAST_COMPLETION, ELEMENT);
  DEADLINE := GET_DEADLINE(ELEMENT, LAST_COMPLETION);
  TARDINESS := COMPLETION - DEADLINE;
  LAST_COMPLETION := COMPLETION;
  INCLUDE_TASK(COMPLETION, STACK);
end GET_TARDINESS;


-- get ancestors of a task

procedure GET_ANCESTOR(TASK_LIST     : in TASK_LINK;
                       STATUS_ID     : in STATUS_VEC;
                       PARENT_MTX    : in LINK_MATRIX;
                       ANCESTOR_VEC  : in out VECTOR;
                       SIZE          : in out INTEGER) is
  STATUS_VALUE : STATUS_VEC(1 .. TASK_SIZE) := STATUS_ID;
  EQUAL        : BOOLEAN                    := FALSE;

  -- void the same task's parent appear repeatly

  procedure COMPARE_EQUAL(ELEMENT   : in INTEGER;
                          TASK_VEC  : in VECTOR;
                          EQUAL     : out BOOLEAN) is

  begin
    for I in 1 .. SIZE loop
      if ELEMENT = TASK_VEC(I) then
        EQUAL := TRUE;
        exit;
      end if;
    end loop;
  end COMPARE_EQUAL;
begin
  if TASK_LIST /= null then
    COMPARE_EQUAL(TASK_LIST.P, ANCESTOR_VEC, EQUAL);
    if not EQUAL and not STATUS_VALUE(TASK_LIST.P) then
      ANCESTOR_VEC(SIZE) := TASK_LIST.P;
      SIZE := SIZE + 1;
      STATUS_VALUE(TASK_LIST.P) := TRUE;
      GET_ANCESTOR(PARENT_MTX(TASK_LIST.P), STATUS_VALUE, PARENT_MTX,
       ANCESTOR_VEC, SIZE);
      GET_ANCESTOR(TASK_LIST.NEXT, STATUS_VALUE, PARENT_MTX,
       ANCESTOR_VEC, SIZE);
    else
      if TASK_LIST.NEXT /= null then
        GET_ANCESTOR(TASK_LIST.NEXT, STATUS_VALUE, PARENT_MTX,
                     ANCESTOR_VEC, SIZE);
      end if;
    end if;
  end if;
end GET_ANCESTOR;

-- estimate the deadline of a task

function ESTIMATE_DEADLINE(ELEMENT     : INTEGER;
                           STATUS_FLAG: STATUS_VEC)return INTEGER is

  OP, INSTANCE, DEADLINE, PHASE_VALUE : INTEGER;
begin
  OP := TASK_OPERATOR(ELEMENT, TASK_VECTOR);
  INSTANCE := TASK_INSTANCE(ELEMENT, TASK_VECTOR);
  if not PHASE_DONE(OP) then
    PHASE_VALUE := GET_UNSCHEDULED_PHASE(ELEMENT, OP);
  else
    PHASE_VALUE := PHASE(OP);
  end if;
  DEADLINE := PERIOD(OP) * INSTANCE + PHASE_VALUE +
   OP_FINISH_WITHIN(OP);
  if DEADLINE > H_B_L then
    DEADLINE := H_B_L;
  end if;
  return DEADLINE;
end ESTIMATE_DEADLINE;

-- caculate the summition of MET of all the unscheduled tasks

function SUM(ANCESTORS : VECTOR;
             SIZE      : INTEGER) return INTEGER is

  SUM_VALUE    : INTEGER                  := 0;
  OP           : INTEGER;
  ANCESTOR_VEC : VECTOR(1 .. TASK_SIZE) := ANCESTORS;
  TASKS        : INTEGER;
begin
  for I in 1 .. SIZE loop
    OP := TASK_OPERATOR(ANCESTOR_VEC(I), TASK_VECTOR);
    SUM_VALUE := SUM_VALUE + OP_MET(OP);
  end loop;
  return SUM_VALUE;
end SUM;
-- caculate the worst tardiness from the scheduled tasks

procedure GET_COST(ELEMENT         : in INTEGER;
                   SEQ_COST        : in out INTEGER;
                   LAST_COMPLETION : in out INTEGER;
                   COST_STACK      : in out TASK_LINK) is
  VALUE : INTEGER;
  TASKS : INTEGER := ELEMENT;

begin
  GET_TARDINESS(TASKS, VALUE, LAST_COMPLETION, STACK);
  if SEQ_COST < VALUE then
    SEQ_COST := VALUE;
  end if;
  INCLUDE_TASK(SEQ_COST, COST_STACK);
end GET_COST;

-- find the least cost of the all unschedule tasks

procedure GET_LEAST_COST(STATUS_V                : in STATUS_VEC;
                         NUMBER, COMPLETION_TIME : in INTEGER;
                         LEAST_VALUE             : out INTEGER) is
  TEMP_VALUE : INTEGER := INTEGER'FIRST;
  E_COST     : INTEGER;

  -- find the estimate value of an unscheduled task

  function ESTIMATE_COST(TASKS, COMPLETION_TIME : INTEGER;
                         STATUS_ID: STATUS_VEC)return INTEGER is

    ANCESTOR_VEC  : VECTOR(1 .. TASK_SIZE) := (others => 0);
    SUM_TIME      : INTEGER;
    DEADLINE_TIME : INTEGER;
    LEAST_COST    : INTEGER;
    SIZE          : INTEGER                := 1;

  begin
    GET_ANCESTOR(PARENT_MTX(TASKS), STATUS_ID, PARENT_MTX,
     ANCESTOR_VEC, SIZE);
    ANCESTOR_VEC(SIZE) := TASKS;
    SUM_TIME := SUM(ANCESTOR_VEC, SIZE);
    DEADLINE_TIME := ESTIMATE_DEADLINE(TASKS, STATUS_ID);
    LEAST_COST := COMPLETION_TIME + SUM_TIME - DEADLINE_TIME;
    return LEAST_COST;
  end ESTIMATE_COST;

begin
  for I in 1 .. TASK_SIZE loop
    if STATUS_V(I) = FALSE then
      E_COST := ESTIMATE_COST(I, LAST_COMPLETION, STATUS_V);
      if TEMP_VALUE < E_COST then
        TEMP_VALUE := E_COST;
      end if;
    end if;
  end loop;
  LEAST_VALUE := TEMP VALUE;
end GET_LEAST_COST;

-- check the last unschedule task is been put into
-- the scheduled sequence or not
```

```
function GET_DONE(STATUS_V : STATUS_VEC;
                  NUMBER   : INTEGER) return BOOLEAN is

  ALL_DONE : BOOLEAN := FALSE;
  COUNT    : INTEGER := 0;

begin
  for I in 1 .. TASK_SIZE loop
    if STATUS_V(I) = TRUE then
      COUNT := COUNT + 1;
    end if;
  end loop;
  ALL_DONE := (COUNT = NUMBER);
  return ALL_DONE;
end GET_DONE;


-- copy a list to a vector

procedure COPY(LIST          : in out TASK_LINK;
               BEST_SEQUENCE : in out VECTOR) is

  SIZE       : INTEGER := 0;
  TASK_VALUE : INTEGER;
begin
  while LIST /= null loop
    SIZE := SIZE + 1;
    GET_TASK(LIST, TASK_VALUE);
    BEST_SEQUENCE(SIZE) := TASK_VALUE;
  end loop;
end COPY;

-- copy a vector of time value to another vector

function GET_TIME(TEMP : in VECTOR) return VECTOR is

  TIME_VEC : VECTOR(1 .. TASK_SIZE);
begin
  for I in 1 .. TASK_SIZE loop
    TIME_VEC(I) := TEMP(I);
  end loop;
  return TIME_VEC;
end GET_TIME;

-- compare two value and return the maximum value

function MAX(LEFT : INTEGER; RIGHT : INTEGER) return INTEGER is

begin
  if LEFT > RIGHT then
    return LEFT;
  else
    return RIGHT;
  end if;
end MAX;

-- get the maximum cost of all the scheduled tasks and all
-- unscheduled tasks compare with  best cost if it is then best
-- cost then do it else bound

procedure BRANCH_BOUND(SEQ_LIST             : in VECTOR;
                       LIST_FLAG, STATUS : in STATUS_VEC;
                       NUMBER               : in INTEGER) is

  LIST        : VECTOR(1 .. TASK_SIZE)      := SEQ_LIST;
  LIST_STATUS : STATUS_VEC(1 .. TASK_SIZE) := LIST_FLAG;

  -- check the task is scheduled(T) or not(F)
  STATUS_V    : STATUS_VEC(1 .. TASK_SIZE) := STATUS;
  TASKS       : INTEGER;
  TASK_VALUE  : INTEGER;

  -- include the no incoming edge node(task) of the graph of
  -- precedence constraints
  QUEUE       : TASK_LINK                  := null;
  TARDINESS   : INTEGER;
  COST        : INTEGER;
  LEAST_COST  : INTEGER;
  MAX_VALUE   : INTEGER;

  -- estimate cost of an unscheduled task
  E_COST      : INTEGER;

  -- the task belong to which operator
  OP_ID       : INTEGER;

  -- the task's instance of an operator
  INSTANCE_ID : INTEGER;

  -- check all the tasks which has been scheduled
  ALL_DONE    : BOOLEAN;

begin
  QUEUE := GET_QUEUE(NUMBER, STATUS_V);

  -- queue equal to null means got a complete sequence
  if QUEUE = null then
    if SEQ_COST < BEST_COST then
      BEST_COST := SEQ_COST;
      BEST_START_TIME := GET_TIME(START_TIME);
      BEST_END_TIME := GET_TIME(END_TIME);
      BEST_SEQUENCE := LIST;
    end if;
    return;
  else
    while QUEUE /= null loop
      GET_TASK(QUEUE, TASKS);
```

```
ADD_TASK(TASKS, LIST, LIST_STATUS);
-- remove the task from the graph
STATUS_V(TASKS) := TRUE;

-- check the task is the last task or not
ALL_DONE := GET_DONE(STATUS_V, NUMBER);
GET_COST(TASKS, SEQ_COST, LAST_COMPLETION, COST_STACK);
if not ALL_DONE then
  GET_LEAST_COST(STATUS_V, NUMBER, LAST_COMPLETION,
   LEAST_COST);
  MAX_VALUE := MAX(SEQ_COST, LEAST_COST);
else
  MAX_VALUE := SEQ_COST;
end if;
if MAX_VALUE < BEST_COST then
  BRANCH_BOUND(LIST, LIST_STATUS, STATUS_V, NUMBER);

  -- best_cost less than or equal to zero means we got the
  -- feasible sequence so quit
  if BEST_COST <= 0 then
    return;
  end if;
end if;

-- remove the task from previous scheduled sequence
REMOVE_TASK(LIST_STATUS);

-- put the task back to the graph(unscheduled)
STATUS_V(TASKS) := FALSE;
if TASKS /= 1 then

  -- remove the completion time of the previous removed task
  REMOVE_STACK(STACK);

  -- get the completion time of the last task
  -- which in the scheduled list
  LAST_COMPLETION := POP_STACK(STACK);

  -- remove the cost of the list which include
  -- the previous removed task
  REMOVE_STACK(COST_STACK);

  -- get the cost of the list which do not include
  -- the previous removed task
  SEQ_COST := POP_STACK(COST_STACK);
end if;
OP_ID := TASK_OPERATOR(TASKS, TASK_VECTOR);
INSTANCE_ID := TASK_INSTANCE(TASKS, TASK_VECTOR);

-- if the instance of the removed task is zero that means
-- the phase of the operator of the task is not in the
-- scheduled list
if INSTANCE_ID = 0 then
  PHASE_DONE(OP_ID) := FALSE;
```

```
      end if;
    end loop;
  end if;
end BRANCH_BOUND;
-- get the matrix of all the tasks' parent
-- in the precedence constraints graph

function GET_PARENT_MATRIX(TASK_PRECEDENCE : MATRIX;
                           TASK_SIZE: INTEGER)return LINK_MATRIX is

  PARENT_MTX : LINK_MATRIX(1 .. TASK_SIZE);

begin
  INITIALIZE_POINTER(PARENT_MTX, TASK_SIZE);
  for J in 1 .. TASK_SIZE loop
    for I in 1 .. TASK_SIZE loop
      if TASK_PRECEDENCE(I, J) = '1' then
        INCLUDE_TASK(I, PARENT_MTX(J));
      end if;
    end loop;
  end loop;
  return PARENT_MTX;
end GET_PARENT_MATRIX;


-- decide the sequence is feasible or optimal and print it out

procedure PRINT_RESULT(BEST_SEQ  : in VECTOR;
                       BEST_COST : in INTEGER) is

begin
  if BEST_COST <= 0 then
    PUT_LINE(" The feasible sequence is ");
    WRITE_VECTOR(TASK_SIZE, BEST_SEQUENCE);
    NEW_LINE(2);
  else
    PUT_LINE(" The optimal sequence is ");
    WRITE_VECTOR(TASK_SIZE, BEST_SEQUENCE);
    NEW_LINE(2);
  end if;
  PUT(" The cost is ");
  PUT(BEST_COST);
  NEW_LINE(2);
  PUT_LINE("         TASK   START TIME  END TIME");
  PUT_LINE("        ------  ---------- ----------");
  for I in 1 .. TASK_SIZE loop
    PUT(BEST_SEQUENCE(I));
    PUT(BEST_START_TIME(BEST_SEQUENCE(I)));
    PUT(BEST_END_TIME(BEST_SEQUENCE(I)));
    NEW_LINE;
  end loop;
end PRINT_RESULT;

-- the body of procedure get_sequence
```

```
        begin
          PARENT_MTX := GET_PARENT_MATRIX(TASK_PRECEDENCE, TASK_SIZE);
          BRANCH_BOUND(LIST, LIST_STATUS, STATUS, TASK_SIZE);
          PRINT_RESULT(BEST_SEQUENCE, BEST_COST);
        end GET_SEQUENCE;
        -- the body of procedure get_task_sequence

      begin
        GENERATE_PRECEDENCE_GRAPH;
        GET_SEQUENCE(TASK_PRECEDENCE, TOTAL_NUMBER_OF_TASKS);
      end GET_TASK_SEQUENCE;


      -- the body of procedure get_schedule

    begin
      GET_DATA;
      H_B_L := EVALUATE_LCM(OPERATORS, PERIOD);
      GET_TOTAL_TASK_NUMBER;
      GET_TASK_SEQUENCE(TOTAL_NUMBER_OF_TASKS);
    end GET_SCHEDULE;


    -- the body of main procedure

  begin
    OPEN(INF, MODE => IN_FILE, NAME => "input_f");
    SKIP_LINE(INF);
    GET(INF, NUMBER_OF_OPERATOR);
    SKIP_LINE(INF);
    GET_SCHEDULE(NUMBER_OF_OPERATOR);
    CLOSE(INF);
  end FIND_SCHEDULE;
```

APPENDIX C
INPUT DATA AND OUTPUT DATA FOR EXAMPLES

INPUT DATA

EXAMPLE 1 :

operator_number :
6

operator	period	met	finish_within	precedence
1	15	2	7	0
2	10	1	8	1
3	15	3	10	1
4	30	4	15	2 3
5	15	2	8	3
6	15	2	12	4 5

EXAMPLE 2:

operator_number :
7

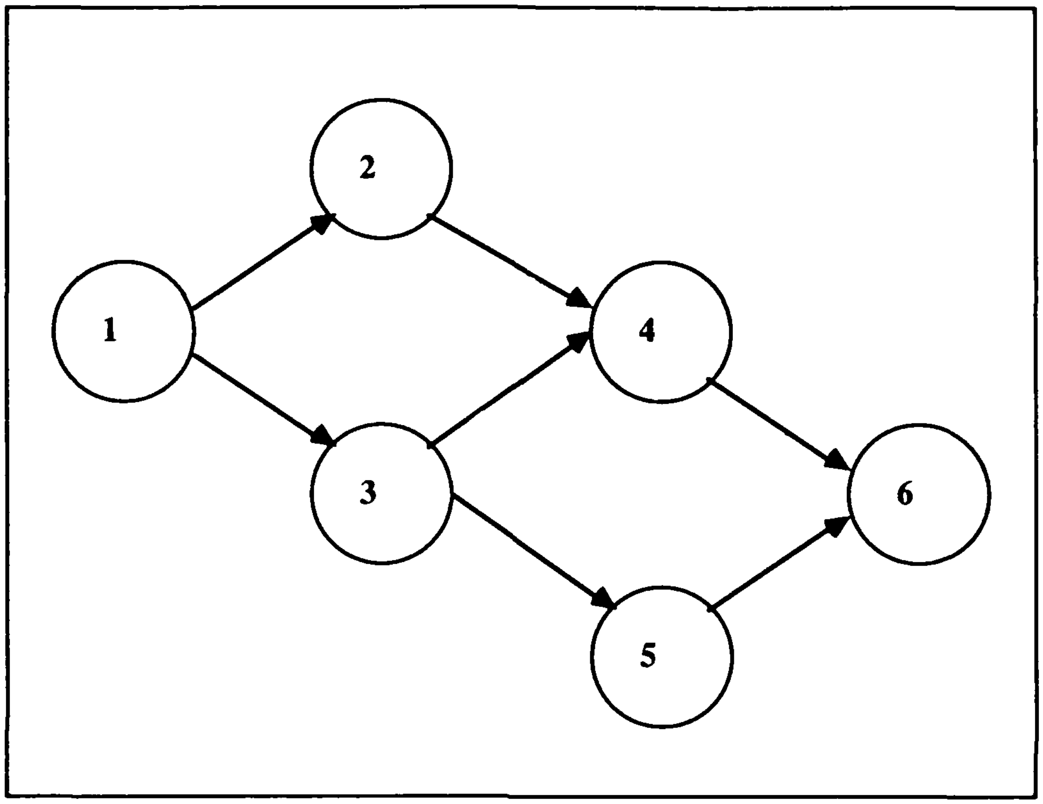
operator	period	met	finish_within	precedence
1	10	2	9	0
2	15	2	10	0
3	15	2	12	0
4	30	2	20	0
5	10	1	8	1 2 3
6	15	1	12	4
7	30	3	18	6

EXAMPLE 3 :

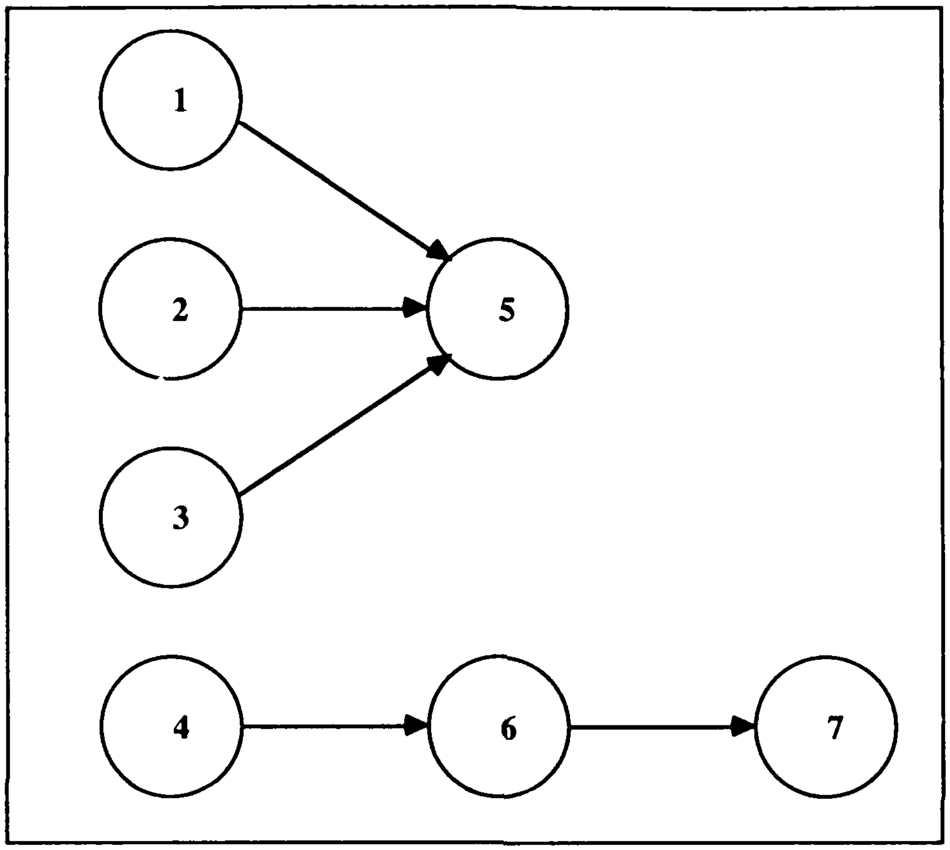
operator_number :
8

operator	period	met	finish_within	precedence
1	10	2	9	0
2	15	1	10	1
3	30	6	15	1
4	30	1	15	1
5	15	3	11	2 3
6	15	1	12	2 4
7	30	1	18	3 4
8	15	1	10	5 6 7

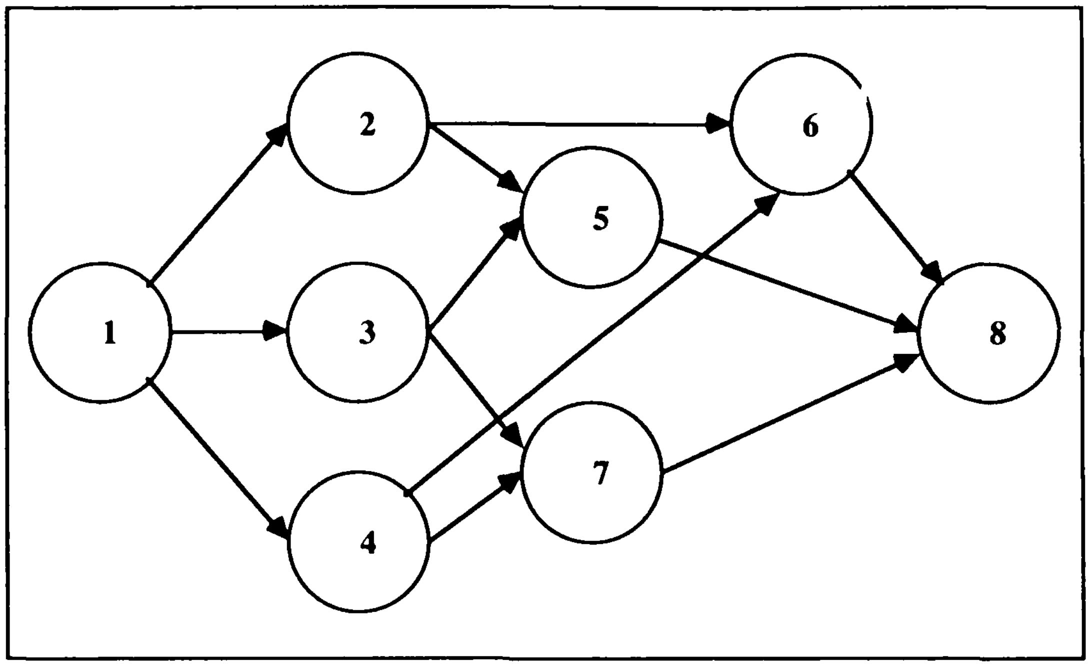
CONSTRAINTS GRAPH



Constraints graph of tasks of example 1



Constraints graph of tasks of example 2



Constraints graph of tasks of example 3

OUTPUT DATA

```
The output data of example 1 :

A. Exhaustive enumeration

 The feasible sequence is
   1   3   6   8   9   4  11   2   5   7  10  12

          TASK   START TIME  END TIME
         ------  ----------  ----------
           1         0           2
           3         2           3
           6         3           6
           8         6          10
           9        10          12
           4        12          13
          11        13          15
           2        15          17
           5        22          23
           7        23          26
          10        26          28
          12        28          30


B. Branch and bound

 The feasible sequence is
   1   3   6   8   9   4  11   2   5   7  10  12

 The cost is          0

          TASK   START TIME  END TIME
         ------  ----------  ----------
           1         0           2
           3         2           3
           6         3           6
           8         6          10
           9        10          12
           4        12          13
          11        13          15
           2        15          17
           5        22          23
           7        23          26
          10        26          28
          12        28          30
```

```
The output data of example 2 :

A. Exhaustive enumeration

 The feasible sequence is
   1    4    6    8    9    2   12    5    3    7   10   14   11   13

         TASK    START TIME  END TIME
        ------  ---------- ----------
          1          0          2
          4          2          4
          6          4          6
          8          6          8
          9          8          9
          2         10         12
         12         12         13
          5         17         19
          3         20         22
          7         22         24
         10         24         25
         14         25         28
         11         28         29
         13         29         30


B. Branch and bound

 The feasible sequence is
   1    4    6    8    9    2   12    5    3    7   10   14   11   13

 The cost is            0

         TASK    START TIME  END TIME
        ------  ---------- ----------
          1          0          2
          4          2          4
          6          4          6
          8          6          8
          9          8          9
          2         10         12
         12         12         13
          5         17         19
          3         20         22
          7         22         24
         10         24         25
         14         25         28
         11         28         29
         13         29         30
```

```
The output data of example 3 :

A. Exhaustive enumeration

 There is no feasible sequence


B. Branch and bound

 The optimal sequence is
   1    4    6    7    8   10   12   13    2    3    5    9   11   14

 The cost is           1

         TASK    START TIME  END TIME
        ------   ----------  ----------
          1          0           2
          4          2           3
          6          3           9
          7          9          10
          8         10          13
         10         13          14
         12         14          15
         13         15          16
          2         16          18
          3         20          22
          5         22          23
          9         25          28
         11         28          29
         14         30          31
```

LIST OF REFERENCE

1. [BB88] Brassard, G., Bratley, P., *Algorithmics*, Prentice-Hall, Englewood Cliffs, NJ., 1988.

2. [Ber89] Berzins, V., Course Notes - CS4500, Naval Postgraduate School, Monterey, CA., 1989.

3. [BFR 71] Bratley, p., Florian, M. and Robillard, P., *Scheduling with Earliest Start and Due Date Constraints*, Naval Research Logistic Quartely, 18, 4, December 1971.

4. [Boo87] Booch, G., *Software Engineering with Ada*, 2nd ed., Benjamin/Cummings Publishing Co., Inc., Menlo Park, CA., 1987.

5. [Bro89] Brookshear, J. G., *Theory of Computation*, Benjamin/Cummings Publishing Co., Inc., Menlo Park, CA., 1989.

6. [BS74] Baker , K. R., Liu, J. W. S., *Sequencing with Due_date and Early Start Times to Minimize Maximum Tardiness*, Naval Research Logistic Quartely, 21, 1974.

7. [BSR88] Biyaabani, S. R., Stankovic, J. A., and Ramamritham, K., *The Integration of Deadline and Criticalness in Hard Real-time Scheduling*, IEE transactions on Software Engineering, 1988.

8. [Cer89] Cervantes, J. J., *An Optimal Static Scheduling Algorithm for Hard Real-time Systems*. M.S. thesis, Naval Postgraduate School, Monterey, CA., Dec. 1989.

9. [Cof76] Coffman, E. G., *Computer and Job-Shop Scheduling Theory*, John Wiley & Sons, Inc., 1976.

10. [Coo83] Cook, S. A., "*An Overview of Computational Complexity*", *Communications of the ACM*, Vol. 26, No. 6, Jun. 1983, pp. 401-408.

11. [CSR87] Cheng, S. C., Stankovic, J. A., Ramamritham, K., *Scheduling Algorithms for Hard Real-time Systems - A Brief Survey*, COINS Technical Report 87-55, Jun. 10, 1987.

12. [DOSE87] Dossey, J. A., Otto, A. D., Spence, L. E., Eynden, C. V., *Discrete Mathematics*, Scott, Foresman and Company, 1987.

13. [Fre82] French, S., *Sequencing and Scheduling*, Ellis Horwood Ltd., 1982.

14. [GGK78] Garey, M. R., Graham, R. L., Johnson, D. S., *Performance Guarantees for Scheduling Algorithms*, Operations Research, Vol. 26, No. 1, Jan.-Feb., 1978, pp. 3-21.

15. [GJ78] Garey, M. R., Johnson, D. S., ""Strong" NP-Completeness Results: Motivation, Examples, and Implications", Journal of the Association for Computing Machinery, Vol. 25, No. 3, Jul. 1978, pp. 499-508.

16. [GJ79] Garey, M. R., Johnson, D. S., *Computers and Intractability; A guide to the Theory of NP-Completeness*, Freeman; San Francisco, 1979.

17. [GM84] Gondran M., Minoux, M., *Graphs and Algorithms*, John Wiley & Sons Ltd., 1984.

18. [HS78] Horowitz, E., Sahni, S., "Fundamentals of Computer Algorithms", Computer Science Press, Rockville, MD., 1978.

19. [Jan88] Janson, D. M., "A Static Scheduler for The Computer Aided Prototyping System: An Implementation Guide", M.S. thesis, Naval Postgraduate School, Monterey, CA., Sep. 1988.

20. [Kil89] Kilic, M., "Static Schedulers for Embedded Real-time Systems". M.S. thesis, Naval Postgraduate School, Monterey, CA., 1989.

21. [LA90] Levi, S., Agrawala, A. K., *Real-Time System Design*, McGraw-Hill Publishing, 1990.

22. [LB88] Luqi, Berzins, V., “Rapidly Prototyping Real-time Systems”, IEEE Software, Sep. 1988, pp. 25-36.

23. [LBY88] Luqi, Berzins, V., Yeh, R. T., “A Prototyping Language for Real-time Software”, IEEE Transactions on Software Engineering, Vol. 14, No. 10, Oct. 1988, pp. 1409-1423.

24. [LK78] Lenstra, J. K., Kan, A. H. G. R., "Complexity of Scheduling under Precedence Constraints", Operations Research, Vol. 26, No. 1, Jan.-Feb. 1978, pp. 22-35.

25. [LK88] Luqi, Ketabchi, M., "A Computer-Aided Prototyping System", IEEE Software, Mar. 1988, pp. 66-72.

26. [Luq88] Luqi, "Knowledge-Based Support for Rapid Software Prototyping", IEEE Expert, Winter 1988, pp. 9-18.

27. [Luq89a] Luqi, "Software Evolution Through Rapid Prototyping", IEEE Computer, May 1989, pp. 13-25.

28. [Luq89b] Luqi, "Rapid Prototyping Languages and Expert Systems", IEEE Expert, Summer 1989, pp. 2-5.

29. [Luq89c] Luqi, "Handling Timing Constraints in Rapid Prototyping", in proceedings of the 22nd Annual Hawaii International Conference on System Science, Kailua-Kona, Hawaii, Jan. 1989.

30. [LW66] Lawler, E. L., Wood, D. E., "Branch-and-Bound Methods: A Survey", Operations Research, Vol. 14, Jul. 1966, pp. 699-719.

31. [Man67] Manacher, G. K., "Production and Stabilization of Real-Time Task Schedules", Journal of the Association for Computing Machinery, Vol. 14, No. 3, Jul. 1967, pp. 439-465.

32. [Man89] Manber, U., *Introduction to Algorithms*, Addison-Wesley Publishing Company Inc., 1989.

33. [Mar88] Marlowe, L., "A Scheduler for Critical Time Constraints", M.S. thesis, Naval Postgraduate School, Monterey, CA., Dec. 1988.

34. [O'He88] O'Hern, J. T., "A Conceptual Level Design for a Static Scheduler for Hard Real-time Systems", M.S. thesis, Naval Postgraduate School, Monterey, CA., Sep. 1988.

35. [Pre87] Pressman, R. S., *Software Engineering*, 2nd ed., McGraw-Hill, Inc., 1987.

36. [SH78] Sahni, S., Horowitz, E., “Combinatorial Problems: Reducibility and Approximation”, Operations Research, Vol. 26, No. 5, Sep.-Oct. 1978, pp. 718-759.

37. [SR88] Stankovic J. A., Ramamritham, K., "Hard Real-Time Systems", IEEE Computer Society Press, Washington, DC., 1988.

38. [Wei77] Weide, B., "A Survey of Analysis Techniques for Discrete Algorithms", Computing Surveys, Vol. 9, No. 4, Dec. 1977, pp. 291-313.

39. [ZRS87] Zhao, W., Ramamritham, K., Stankovic, J. A., "Preemptive Scheduling Under Time and Resource Constraints", IEEE Transactions on Computers, Vol. C-36, No. 8, Aug. 1987, pp. 949-960.

40. [CR75] Chandy, K. M., Reynolds, P. F., "Scheduling partially ordered tasks with probabilistic execution times", in Proceedings of the 5th Symposium on Operating Systems Principles, 1975.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943	2
Director of Research Administration Code 012 Naval Postgraduate School Monterey, CA 93943	1
Chairman, Code 52 Computer Science Department Naval Postgraduate School Monterey, CA 93943	1
Office of Naval Research 800 N. Quincy Street Arlington, VA 22217-5000	1
Center for Naval Analysis 4401 Ford Avenue Alexandria, VA 22302-0268	1
National Science Foundation ATTN: Dr. K. C. Tai Division of Computer and Computation Research Washington, D.C. 20550	1
Office of the Chief of Naval Operations Code OP-941 Washington, D.C. 20350-2000	1
Office of the Chief of Naval Operations Dr. John R. Davis, Chief Scientist Code OP-094H Washington, D.C. 20350-2000	1

Office of the Chief of Naval Operations 1
Code OP-945
Washington, D.C. 20350-2000

Office of the Chief of Naval Operations 1
ATTN: CDR R. V. Gragg
Code OP-942F4
Washington, D.C. 20350-2000

Commander 2
Naval Sea Systems Command
ATTN: LCDR Scott Kelly
Code 06D3131
Washington, D.C. 20362-5101

Commander 2
Naval Telecommunications Command
Naval Telecommunications Command Headquarters
4401 Massachusetts Avenue NW
Washington, D.C. 20390-5290

Commander 1
Naval Data Automation Command
Washington Navy Yard
Washington, D.C. 20374-1662

Commander 2
Space and Naval Warfare Systems Command
ATTN: CAPT John Gauss
PMW-162
Washington, D.C. 20363-5100

Assistant Secretary of the Navy 1
Research, Development and Acquisitions
Washington, D.C. 20350-1000

Dr. Lui Sha 1
Carnegie Mellon University
Software Engineering Institute
Department of Computer Science
Pittsburgh, PA 15260

Commanding Officer Code 5150 Naval Research Laboratory Washington, D.C. 20375-5000	1
Defense Advanced Research Projects Agency (DARPA) Director, Naval Technology Office 1400 Wilson Boulevard Arlington, VA 2209-2308	1
Defense Advanced Research Projects Agency (DARPA) Director, Prototype Projects Office 1400 Wilson Boulevard Arlington, VA 2209-2308	1
Defense Advanced Research Projects Agency (DARPA) Director, Tactical Technology Office 1400 Wilson Boulevard Arlington, VA 2209-2308	1
Dr. R. M. Carroll (OP-01B2) Chief of Naval Operations1 Washington, DC 20350	1
Dr. Robert M. Balzer USC-Information Sciences Institute 4676 Admiralty Way Suite 1001 Marina del Ray, CA 90292-6695	1
Dr. Ted Lewis Editor-in-Chief, IEEE Software Oregon State University Computer Science Department Corvallis, OR 97331	1
Dr. R. T. Yeh International Software Systems Inc. 12710 Research Boulevard, Suite 301 Austin, TX 78759	1

Dr. C. Green Kestrel Institute 1801 Page Mill Road Palo Alto, CA 94304	1
Prof. D. Berry Department of Computer Science University of California Los Angelas, CA 90024	1
Director, Naval Telecommunications System Integration Center NAVCOMMUNIT Washington Washington, D.C. 20363-5110	1
Dr. Knudsen Code PD50 Space and Naval Warfare Systems Command Washington, D.C. 20363-5110	1
Ada Joint Program Office OUSDRE(R&AT) The Pentagon Washington, D.C. 23030	1
CAPT A. Thompson Naval Sea Systems Command National Center #2, Suite 7N06 Washington, D.C. 22202	1
Dr. Peter Ng New Jersey Institute of Technology Computer Science Department Newark, NJ 07102	1
Dr. Van Tilborg Office of Naval Research Computer Science Division, Code 1133 800 N. Quincy Street Arlington, VA 22217-5000	1
Dr. R. Wachter Office of Naval Research Computer Science Division, Code 1133 800 N. Quincy Street Arlington, VA 22217-5000	1

Dr. J. Smith, Code 1211 1
Office of Naval Research1
Applied Mathematics and Computer Science
800 N. Quincy Street
Arlington, VA 22217-5000

Mr. William E. Rzepka 1
U.S. Air Force Systems Command
Rome Air Development Center
RADC/COE
Griffis Air Force Base, NY 13441-5700

Dr. C.V. Ramamoorthy 1
University of California at Berkeley
Department of Electrical Engineering and Computer Science
Computer Science Division
Berkeley, CA 90024

Dr. Nancy Levenson 1
University of California at Irvine
Department of Computer and Information Science
Irvine, CA 92717

Mr. George Roberson 1
Fleet Combat Direction Systems Support Activity
San Diego, CA 92147-5081

Mr. William Hanley 1
Fleet Combat Direction Systems Support Activity
San Diego, CA 92147-5081

Dr. Earl Chavis (OP-162) 1
Chief of Naval Operations
Washington, DC 20350

Dr. Alan Hevner 1
University of Maryland
College of Business Management
Tydings Hall, Room 0137
College Park, MD 20742

Dr. Al Mok University of Texas at Austin Computer Science Department Austin, TX 78712	1
George Sumiall US Army Headquarters CECOM AMSEL-RD-SE-AST-SE Fort Monmouth, NJ 07703-5000	1
Mr. Joel Trimble 1211 South Fern Street, C107 Arlington, VA 22202	1
Linwood Sutton Code 423 Naval Ocean Systems Center San Diego, CA 92152-5000	1
Dr. Sherman Gee Code 221 Office of Naval Technology 200 N. Quincy St. Arlington, VA 22217	1
Dr. Mark Kellner Carnegie-Mellon University Software Engineering Institute Pittsburgh, PA 15213	1
Dr. Luqi Code 52Lq Computer Science Department Naval Postgraduate School Monterey, CA 93943	1
Dr. Valdis Berzins Code 52Bz Computer Science Department Naval Postgraduate School Monterey, CA 93943	30

Dr. Man-Tak Shing 1
Code 52Sh
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Commanding Officer 1
Naval Command and Control Systems Command
Wibbelhof St. 3
2340 Wilhelmshaven
West Germany

Commander 1
Naval Ocean Systems Center
Code 43
San Diego, CA 92152-5000
ATTN: CDR Wayne Duke

Commander 1
Naval Ocean Systems Center
Code 44
San Diego, CA 92152-5000
ATTN: Dr. Glenn Osga

Dr. Chuck Hutchins 1
Code 55Hu
Operations Research Department
Naval Postgraduate School
Monterey, CA 93943

Dr. Gary Poock 1
Code 55Pk
Operations Research Department
Naval Postgraduate School
Monterey, CA 93943

LCDR James A. Seveney 2
8507 Shirley Woods Ct.
Lorton, VA 22079

Commanding Officer 2
1st Destroyer Squadron
Mecklenburger St. 50 A I

2300 Kiel West Germany ATTN: LCDR Guenter P. Steinberg	1
CDR. Bao-Hua Fan 4F, No 2-47 Tsoying Ta Road Tsoying Kaoshiung, Taiwan R.O.C	5